EVALUATION OF

NETWORK RELIABILITY

CALCULATION METHODS

(White Paper)

L. E. Miller, December 2004

Extracted from

L. E. Miller, J. J. Kelleher, and L. Wong, "Assessment of Network Reliability Calculation Methods," J. S. Lee Associates, Inc. report JC-2097-FF under contract DAAL02-92-C-0045, January 1993.

BLANK PAGE

EVALUATION OF NETWORK RELIABILITY CALCULATION METHODS

ABSTRACT

This report is extracted from [13], a 1993 report with limited distribution. Some abridgement and all references to specific military systems or applications have been removed to produce this extract.

The majority of this report is concerned with the development of improved methods for the truncation (*i. e.*, early termination) of algorithms for calculating network reliability that are of the "equivalent link" class, which are based on partitioning the set of possible network success and failure events in an efficient manner. Generally the truncation methods involve detecting when lower and upper bounds on the reliability have converged to a certain degree of closeness. For the convenience of the reader, a review of the mathematical notation associated with this subject is first presented, followed by descriptions of algorithmic approaches and of issues related to algorithm truncation.

In addition to the development of improvements to equivalent-links algorithms, the report summarizes evaluations of network reliability algorithms that are based on application of the factoring theorem in conjunction with techniques for simplifying or reducing the network topology in such a way that accelerates the computations, including one recently published algorithm that is a hybrid of factoring/reduction and of partitioning concepts. As part of this work, advanced techniques were developed for making the reduction class of algorithm compute approximations that are upper and lower bounds on the reliability, thereby shortening the computation time further, and these improvements are documented in this report as well.

On the basis of the numerical results produced by this study, the following was concluded in [13]:

- The concept of using cutsets instead of paths when calculating the upper bound on s-t reliability is effective in improving the convergence of the upper bound, but only for smaller networks or for large networks with relatively high link reliabilities. For larger networks, at some point, usually when the link reliabilities are not high, the events generated by processing cutset failure events become both numerous and quite small in probability, requiring excessive computational time.

- While a program combining cutsets and paths performed reasonably well, of the partitioning class of algorithms, the original equivalent-links algorithm (ELA) (with pathfinding only) seems to work the best in terms of both speed and accuracy.

- A new algorithm combining partitioning and network reduction techniques, modified to include adaptive probability thresholds in order to calculate bounds, shows potential for calculating s-t reliabilities faster than the ELA. However, this new approach does not include a convenient mechanism for stipulating a limit on the lengths of paths, nor one for trading off the prescribed accuracy with the run time, which are important considerations in modeling actual networks.

EVALUATION OF NETWORK RELIABILITY CALCULATION METHODS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

BLANK PAGE

1. INTRODUCTION

This report documents work with the objective of investigating methods for efficient calculation and/or approximation of network reliability. The majority of this report is concerned with the development of improved methods for truncating algorithms for calculating network reliability. For the convenience of the reader, a review of the mathematical notation associated with this subject is first presented, followed by descriptions of algorithmic approaches and of issues related to algorithm truncation.

1.1 BACKGROUND AND NOTATION

1.1.1 Network Representation

The networks of interest are tactical radio networks, represented by directed graphs with imperfect vertices (nodes) and edges (links). The reliabilities of the N nodes are characterized by the numbers $\{\alpha_i\,;\ i = 1, 2,\ldots, N\}$ and the reliabilities of the links, by the numbers $\{\beta_i\,;\ i = N+1,\ N+2,\ldots,\ N+M\}$; alternatively, when convenient the link reliabilities are denoted by $\{\beta_{ij}\,;\ i, j = 1, 2,\ldots, N\}$ although for some tactical radio systems not all of the $N(N-1)$ links $\{i \rightarrow j\}$ are present. For example, in Figure 1-1 a grid network is shown that has 9 nodes and 24 directed links; the nodes as elements are indexed by the numbers 1-9, and the links are indexed by the numbers 10-33. Also shown in Figure 1-1 is an $N \times N = 9 \times 9$ matrix $\boldsymbol{G}$ indicating which pairs of nodes are directly connected and by what links. The link numbers are assigned in the order in which they would be read if the entries in the matrix formed a paragraph of text.

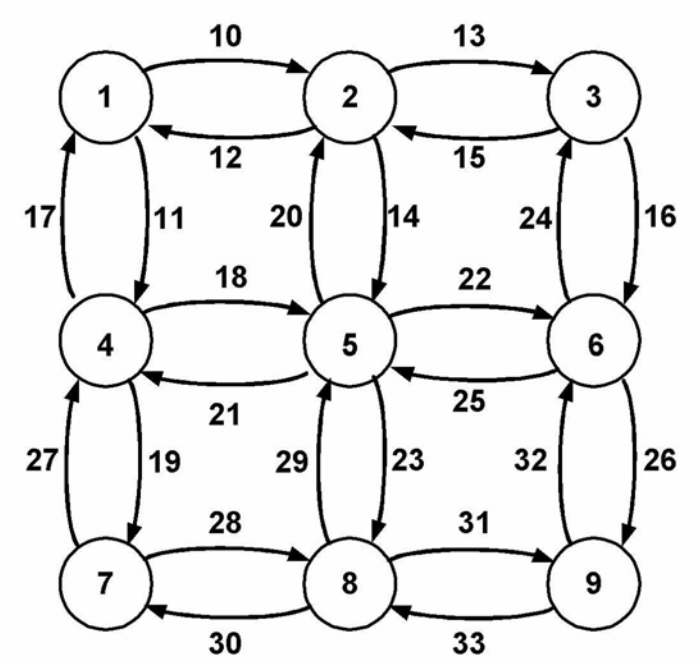
Paths. Each path P connecting particular nodes in the network may be thought of as the set of nodes and links used, which may be represented by a $1 \times (N+M)$ vector

$$\underline{\mathrm{P}} = [p_1\ \ p_2\ \ p_3\ \ \cdots\ \ p_{N+M}] \tag{1-1a}$$

whose entries are

$$p_i = \begin{cases} 1, & i \in P; \\ 0, & \text{otherwise.} \end{cases} \tag{1-1b}$$

For example, one path from node 1 to node 3 in the network of Figure 1-1 can be described by the set of elements $P = (1, 2, 3, 10, 13)$ including nodes 1, 2, and 3 and links 10 and 13. The corresponding 33-element vector $\underline{\mathrm{P}}$ is



Nodes:

$\Pr\{n_i\} = \alpha_i$

Links:

$\Pr\{l_k\} = \beta_k$

Alternate link notation:

$l_{10} \equiv l_{1,2}$ etc.

(a) Weighted Graph

$$\boldsymbol{G} = \begin{bmatrix} 0 & 10 & 0 & 11 & 0 & 0 & 0 & 0 & 0 \\ 12 & 0 & 13 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 15 & 0 & 0 & 0 & 16 & 0 & 0 & 0 \\ 17 & 0 & 0 & 0 & 18 & 0 & 19 & 0 & 0 \\ 0 & 20 & 0 & 21 & 0 & 22 & 0 & 23 & 0 \\ 0 & 0 & 24 & 0 & 25 & 0 & 0 & 0 & 26 \\ 0 & 0 & 0 & 27 & 0 & 0 & 0 & 28 & 0 \\ 0 & 0 & 0 & 0 & 29 & 0 & 30 & 0 & 31 \\ 0 & 0 & 0 & 0 & 0 & 32 & 0 & 33 & 0 \end{bmatrix}.$$

(b) Combined Node Adjacency and Link Identification Matrix

FIGURE 1-1 EXAMPLE NETWORK REPRESENTATION

$$\underline{\mathrm{P}} = [\,1\;1\;1\;0\;0\;0\;0\;0\;0\;1\;0\;0\;1\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0\,]\,. \tag{1-1c}$$

Events. In a particular realization of the probabilistic network, a given element may or may not be operable. Let the indicator variable X_i denote the operability of the ith element, where

$$X_i = \begin{cases} 1, & \text{element } i \text{ operable;} \\ -1, & \text{element } i \text{ not operable.} \end{cases} \tag{1-2}$$

An *elementary event* (e) is defined to be one of the 2^{N+M} particular realizations of the network, i.e. a specification of the operational status of each weighted element. The elementary event may be represented as a $1 \times (N + M)$ vector,

$$\underline{\mathrm{e}} = [X_1\; X_2 \cdots X_{N+M}]\,, \tag{1-3}$$

where all the entries equal +1 or -1. Another way to describe an elementary event is as the logical expression formed by the intersection of logical variables (or their complements) representing the status of each of the $N + M$ elements. Note that elementary events are mutually exclusive (disjoint).

A *general event* E is the union of certain elementary events e_1, e_2 ,..., e_n. The general event E is represented by a $1 \times (N + M)$ vector

$$\underline{\mathrm{E}} = [\xi_1\; \xi_2 \cdots \xi_{N+M}] \tag{1-4}$$

where

$$\xi_i = \begin{cases} 1, & X_i = 1 \text{ for all } e_j \in E,\ j = 1, 2, \ldots, n; \\ -1, & X_i = -1 \text{ for all } e_j \in E,\ j = 1, 2, \ldots, n; \\ 0, & \text{otherwise.} \end{cases} \tag{1-5}$$

In other words, if the status of element i is the same for all $e_j \in E$, then X_i encodes that status (± 1); if the status of element i is $+1$ in some $e_j \in E$ and -1 for some other $e_i \in E$, then $X_i = 0$.

A vector $\underline{\mathrm{E}}$ does not uniquely code all possible events. Of all events with the same vector representation, we define a *full event* as the one with the greatest number of elementary events. For this analysis, only full events need be considered; hence an event E with z zeros in its representation is assumed to include the union of 2^z elementary events. Note that a path P may be interpreted as an event, and the representation of P given in (1-1c) properly represents the event.

The universal event I is defined as the union of all elementary events. Note that the event I is represented in the above notation by the all-zeros vector.

For a more compact notation, E can also be written as a list of the nonzero entries in $\underline{\mathrm{E}}$, with an overbar denoting which entries are -1. For example, for

$$\underline{\mathrm{E}} = [1 \;\; 0 \;\; 0 \; -1 \;\; 1 \;\; 0 \cdots 0] \tag{1-6a}$$

where the ellipsis represents all zeros, can also be written

$$E = \{1 \;\; \overline{4} \;\; 5\} \tag{1-6b}$$

or alternatively

$$E = (1, \overline{4}, 5). \tag{1-6c}$$

In Boolean logic notation, the “full" events as described above can be considered to be the joint occurrence of logic variables or their complements. For example, the event specified by (1-6a) requires that the events “element 1 is operative" and “element 5 is operative" be TRUE, and that the event “element 4 is operative" be FALSE. The status of all other elements is irrelevant (the DON'T CARE condition). Thus in Boolean notation, using x to denote a DON'T CARE status for a particular network element, the equivalent to (1-6a) would be

$$\underline{\mathrm{E}} = 1\, x\, x\, 0\, 1\, x \cdots x\,. \tag{1-6d}$$

The intersection of two events A and B with vectors $\underline{\mathrm{A}} = [a_1 \;\; a_2 \;\; \cdots \;\; a_{N+M}]$ and $\underline{\mathrm{B}} = [b_1 \;\; b_2 \;\; \cdots \;\; b_{N+M}]$ is defined as follows. If for any i, both a_i and b_i are nonzero (± 1) but do not agree, then their intersection does not exist, that is, $E = A \cap B = \emptyset$, the empty set; otherwise, each term of the vector $\underline{\mathrm{E}}$ for $E = A \cap B$ is given by the rule

$$e_i = \begin{cases} a_i, & a_i \neq 0 \text{ and } b_i = 0 \text{ or } b_i = a_i; \\ b_i, & b_i \neq 0 \text{ and } a_i = 0 \text{ or } a_i = b_i; \\ 0, & a_i = b_i = 0. \end{cases} \tag{1-7}$$

Note that if $a_i = b_i \neq 0$, then both of the first two cases of (1-7) yield the same result and there is no ambiguity in the definition (remember that if a_i and b_i are both nonzero and unequal, then $E = A \cap B = \emptyset$).

Success and Failure Collections. Two classes of events are of particular interest in analyzing the network. For a given node pair (s, t), an event $S_j(s, t)$ is a success event if for every elementary event $e_i \in S_j$ the realization of the network corresponding to e_i contains at least one s-t path. An event $F_j(s, t)$ is a failure event if for every $e_i \in F_j$ the realization of the network corresponding to e_i contains no s-t paths.

Further, two *collections* of events may be defined. For a given node pair (s, t), a disjoint exhaustive success collection $\mathcal{S}(s, t)$ is a collection of N_s disjoint success events,

$$\mathcal{S} = \{S_1, S_2, \ldots, S_{N_s}\} \tag{1-8}$$

such that if the network realization corresponding to the elementary event $\underline{e}$ contains an s-t path, then $e \in S_j$ for one and only one $S_j \in \mathcal{S}$. Furthermore, there are no elementary events which produce a successful s-t path that are not included in a success event in $\mathcal{S}$. Similarly, a disjoint exhaustive failure collection $\mathcal{F}(s, t)$ is a collection

$$\mathcal{F} = \{F_1, F_2, \ldots, F_{N_f}\} \tag{1-9}$$

such that if the network realization corresponding to e contains no s-t paths, then $e \in F_j$ for one and only one $F_j \in \mathcal{F}$, and there are no elementary events failing to produce an s-t path that are not included in one of the failure events in $\mathcal{F}$. The significance of these exhaustive collections of events is that the s-t reliability, γ_{st}, is the sum of the probabilities of the N_s success events, which also equals one minus the sum of the probabilities of the N_f failure events.

Partitioning of the complement of an event. In order to discuss an algorithm for generating $\mathcal{S}$ and $\mathcal{F}$, we need one more result. Let X be an event specified by a vector with K nonzero elements, and further, without loss of generality, let these elements be elements 1, 2, …, K. Then the complement of X, denoted by $\overline{X}$, can be partitioned as

$$\overline{X} = \{\overline{1}\} \; + \; \{1 \;\; \overline{2}\} \; + \; \{1 \;\; 2 \;\; \overline{3}\} \; + \; \cdots \; + \; \{1 \;\; 2 \;\; 3 \cdots \overline{K}\} \tag{1-10a}$$

where

$$\{\overline{1}\} = [-1 \;\; 0 \;\; 0 \cdots 0], \qquad \{1 \;\; \overline{2}\} = [\; 1 \; -1 \;\; 0 \cdots 0], \text{ etc.} \tag{1-10b}$$

The terms on the right hand side of (1-10a) represent disjoint sets, and hence $\Pr\{\overline{X}\}$ is the sum of their individual probabilities.

1.1.2 The Equivalent Links Algorithm

Fundamentally, the ELA or equivalent-links algorithm ([1], [2]) is a method for generating *disjoint* success and failure events for a given node pair, plus a method of accounting for node failures efficiently. The event-generation portion of the ELA is related to Dotson's method [3]. With reference to Figure 1-2, the way that the ELA works may be explained as follows:

(a) *Initialization.* Events are dimensioned $1 \times M$, as if there were no possibility of node failures. The particular node pair (s, t) is specified, and the success and failure collections $\mathcal{S}$ and $\mathcal{F}$ (lists of success and failure event vectors stored in computer

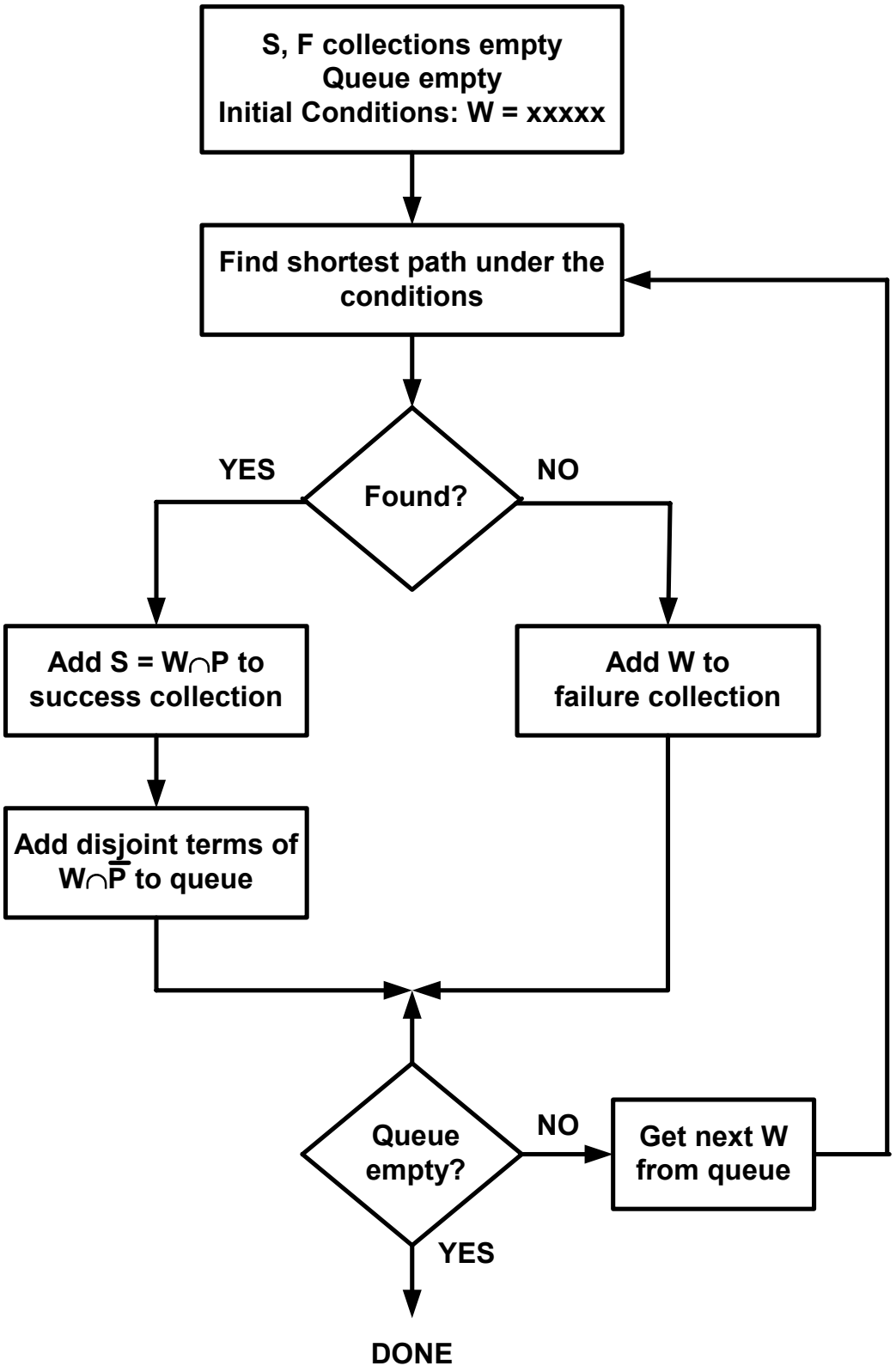


FIGURE 1-2 FLOW DIAGRAM FOR THE EQUIVALENT-LINKS ALGORITHM

memory), are emptied. In addition to $\mathcal{S}$ and $\mathcal{F}$ there is a first-in-first-out queue, denoted $\mathcal{W}$, whose purpose is to buffer network events that are to be tested to see whether they are success or failure events; this queue is initialized to contain one event, the universal event. Recall that the universal event is the set which is union of all possible combinations of network element conditions; its vector representation contains only zeros. In Boolean notation, the logical function representing the universal event is

$$\underline{\mathrm{I}} = x\,x\,x\,x\,x\,x\,x\cdots x\,x\,x\,x; \tag{1-11}$$

all the logic variables are specified to be in the DON'T CARE state.

(b) *Event classification.* The algorithm takes the first event W from the queue and uses a pathfinding algorithm to determine if there is a successful connection from s to t. If no path can be found, the event W is declared to be a failure and it is added to the failure collection. If a path P is found, then the event $W \cap P$ is declared to be a success event and is added to the success collection.

(c) *Disjoint event generation.* After a success event $W \cap P$ has been found, the disjoint events generated by the terms of $W \cap \overline{P}$ are put into the queue. As shown in (1-10), if there are n hops in the path P, then there are n disjoint terms of $\overline{P}$. These give rise to as many as n "next events" to be put into the queue for classification; quite often there are fewer than n next events because the intersection of a term or terms of $\overline{P}$ with W is empty. It is shown in Appendix A.2 of [1] that it is desirable to order the elements of P, and consequently of $\overline{P}$, in the sequence of links traversed from s to t in order to avoid generating events for which successive nodes in the path are operating but are unconnected because of link failures.

(d) *Termination.* The algorithm terminates by itself when the queue becomes empty, that is, when there are no more next events to be classified as either successes or failures. It is possible also to terminate the algorithm early (methods for doing so are a subject of the investigations summarized in this report).

In Appendix A.1 of this report, an example of the operation of the ELA in finding success and failure events for a particular network model is given in detail.

The success and failure vectors $\{\underline{\mathrm{S}}_i\}$ and $\{\underline{\mathrm{F}}_i\}$ which represent the contents of the sets $\mathcal{S} = \{S_i\}$ and $\mathcal{F}\{F_i\}$ in the implementation of the ELA have the dimension $1 \times M$, where M is the number of links (edges). The M components of a particular $\underline{\mathrm{S}}$ vector, for example, take the values 1, if the network element is required to be working; -1, if the network element is required not to be working; or 0, if it doesn't matter what the status of

the element is. Each S corresponds to a term in the polynomial expression for the reliability of the connection between the given source and terminal nodes in the case that all the nodes are perfectly reliable ($\alpha_i = 1.0$ for $i = 1$ to N); the 1's indicate those links whose probabilities of working (reliabilities) β are factors in the term, and the -1's indicate those whose complementary probabilities $1 - \beta$ are factors. The total expression for the *s-t* reliability when the nodes do not fail is then

$$\begin{aligned}\gamma_{st} &= \gamma_{st}(\beta_{N+1}\,, \beta_{N+2}\,, \ldots, \beta_{N+M}) \\ &= \sum_{j=1}^{N_s} \Pr\{S_j(s, t)\} \qquad \text{(1-12a)} \\ &= \sum_{j=1}^{N_s} \prod_r \{\beta_r\colon \mathrm{S}_j(r; s, t) = 1\} \prod_r \{1 - \beta_r\colon \mathrm{S}_j(r; s, t) = -1\}, \qquad \text{(1-12b)}\end{aligned}$$

in which $\mathrm{S}_j(r; s, t)$ denotes the rth component of the vector representing $S_j(s, t)$.

To account for node failures, the node reliabilities are “backfitted" into the expression according to the concept of “equivalent links," in which the network links are thought of as being lumped together with the nodes on which they terminate; to the failure set generated by this method must be added the failure F_0 corresponding to the failure of the source node, s, since none of the links used for the (s, t) connection terminate on s. The formula for factoring in the node reliabilities [2] is based the observation that

$$\Pr\{S_j(s, t)\} = \alpha_s \prod_n \Pr\{S_{jn}(s, t)\}, \qquad \text{(1-13)}$$

where *n* indexes the nodes and $S_{jn}(s, t)$ is the subset of $S_j(s, t)$ which lists the links terminating on node *n* that must be either working or not working. Let N_{jn} denote the number of links required not to be working and K_{jn} the number required to be working, as determined by the algorithm described above. Then the equivalent-link formulas are, assuming that the links are renumbered for convenience [2],

$$\begin{aligned}\Pr\{S_{jn}(s, t)\} &= 1, && N_{jn} = K_{jn} = 0; && \text{(1-14a)} \\ &= \alpha_n \prod_{r=1}^{K_{jn}} \beta_r\,, && N_{jn} = 0,\ K_{jn} > 0; && \text{(1-14b)} \\ &= 1 - \alpha_n + \alpha_n \prod_{r=1}^{N_{jn}} (1 - \beta_r)\,, && N_{jn} > 0,\ K_{jn} = 0; && \text{(1-14c)} \\ &= \alpha_n \prod_{r=1}^{K_{jn}} \beta_r \prod_{r=K_{jn}+1}^{N_{jn}+K_{jn}} (1 - \beta_r), && N_{jn} > 0,\ K_{jn} > 0. && \text{(1-14d)}\end{aligned}$$

In Appendix A.2 of this report, the formulas in (1-14) are applied to success events found by the ELA in a particular case.

1.1.3 Connectivities in Terms of Event Probabilities

As already noted, the disjoint success and failure events generated by the ELA relate directly to the connectivity between pairs of network nodes. The probability that the flood search succeeds in finding a path from origin s to terminal t (or the s-t reliability), in terms of a disjoint exhaustive link success events collection $\mathcal{S} = \{S_1, S_2, \ldots, S_{N_s}\}$, can be written as

$$\gamma_{st} = \sum_{i=1}^{N_s} \Pr\{S_i\} \tag{1-15a}$$

where the link states associated with the event S_i are represented by the vector

$$\underline{\mathrm{S}}_i = [X_{i1} \;\; X_{i2} \;\; \cdots \;\; X_{iM}]\,, \tag{1-15b}$$

and the probability of the events is computed using (1-14). An alternative formulation in terms of an exhaustive link failure event collection $\mathcal{F} = \{F_1, F_2, \ldots, F_{N_f}\}$ is

$$\gamma_{st} = 1 - \sum_{i=1}^{N_f} \Pr\{F_i\} - \Pr\{F_0\} = \alpha_s - \sum_{i=1}^{N_f} \Pr\{F_i\} \tag{1-16a}$$

where F_0 is the source node failure event and the link states associated with the event F_i are represented by the vector

$$\underline{\mathrm{F}}_i \;\; = [X_{i1} \;\; X_{i2} \cdots X_{iM}]\,. \tag{1-16b}$$

Furthermore, suppose exhaustive collections $\mathcal{S}$ and $\mathcal{F}$ cannot be found due to computer time and/or memory limitations. If partial collections $\mathcal{S}' = \{S_1, \ldots, S_K\}$, $K < N_s$ and $\mathcal{F}' = \{F_1, \ldots, F_L\}$, $L < N_f$ can be found, there are the bounds

$$\sum_{i=1}^{K} \Pr\{S_i\} \le \gamma_{st} \le \alpha_s - \sum_{i=1}^{L} \Pr\{F_i\}, \tag{1-17}$$

since all the terms of the summations in (1-15a) and (1-16a) are positive.

1.2 ELA2: A CUTSET APPROACH

It was noted in [1] that, because it is based on pathfinding, the ELA tends to find the majority of success events earlier in its operation, and the majority of failure events, later. A consequence of this behavior is that the convergence of the bounds in (1-17) is uneven, requiring the algorithm to be run longer for the same accuracy than it would have to if the

bounds converged at the same rate. In [4], the concept of basing the event generation on finding cutsets instead of paths is put forth with the idea of finding failure events earlier, thus improving the convergence of the upper bound and perhaps allowing for an earlier termination of the algorithm with the same accuracy. For convenience, the equivalent-links algorithm modified to use a cutset-finding approach shall be referred to as "ELA2."

1.2.1 Operation of ELA2

A flow diagram for the ELA2 is shown in Figure 1-3. The way that the ELA2 works may be explained as follows:

(a) *Initialization.* Events are dimensioned $1 \times M$, as if there were no possibility of node failures. The particular node pair (s, t) is specified, and the success and failure collections $\mathcal{S}$ and $\mathcal{F}$ are emptied. In addition to $\mathcal{S}$ and $\mathcal{F}$ there is a first-in-first-out queue, denoted $\mathcal{W}$, whose purpose is to buffer network events that are to be tested to see whether they are success or failure events; this queue is initialized to contain one event, the universal event.

(b) *Event classification.* The algorithm takes the first event W from the queue and uses a cutset-finding algorithm to determine if there is a set of link outages that precludes a successful connection from s to t. If no cutset can be found, it is because among the links specified as UP in the event W, one or more form an s-t path; in the case of no cutset, W is declared to be a success and it is added to the success collection without further processing. If a cutset

$$C = \{\bar{l}_a\,, \bar{l}_b\,, \bar{l}_c\,, \dots\} \tag{1-18}$$

is found, then the event $W \cap C$ is declared to be a failure event and is added to the failure collection.

(c) *Disjoint event generation.* After a failure event $W \cap C$ has been found, the disjoint events generated by the terms of $W \cap \overline{C}$ are put into the queue. If there are n link outages in the cutset C, then there are n disjoint terms of $\overline{C}$. These give rise to n "next events" to be put into the queue for classification, of the form

$$W \cap \overline{C} = W \cap (l_a \cup \bar{l}_a l_b \cup \bar{l}_a \bar{l}_b l_c \cup \cdots). \tag{1-19}$$

(d) *Termination.* The algorithm terminates by itself when the queue becomes empty, that is, when there are no more next events to be classified as either successes or failures. The algorithm can also be terminated early according to a stopping criterion.

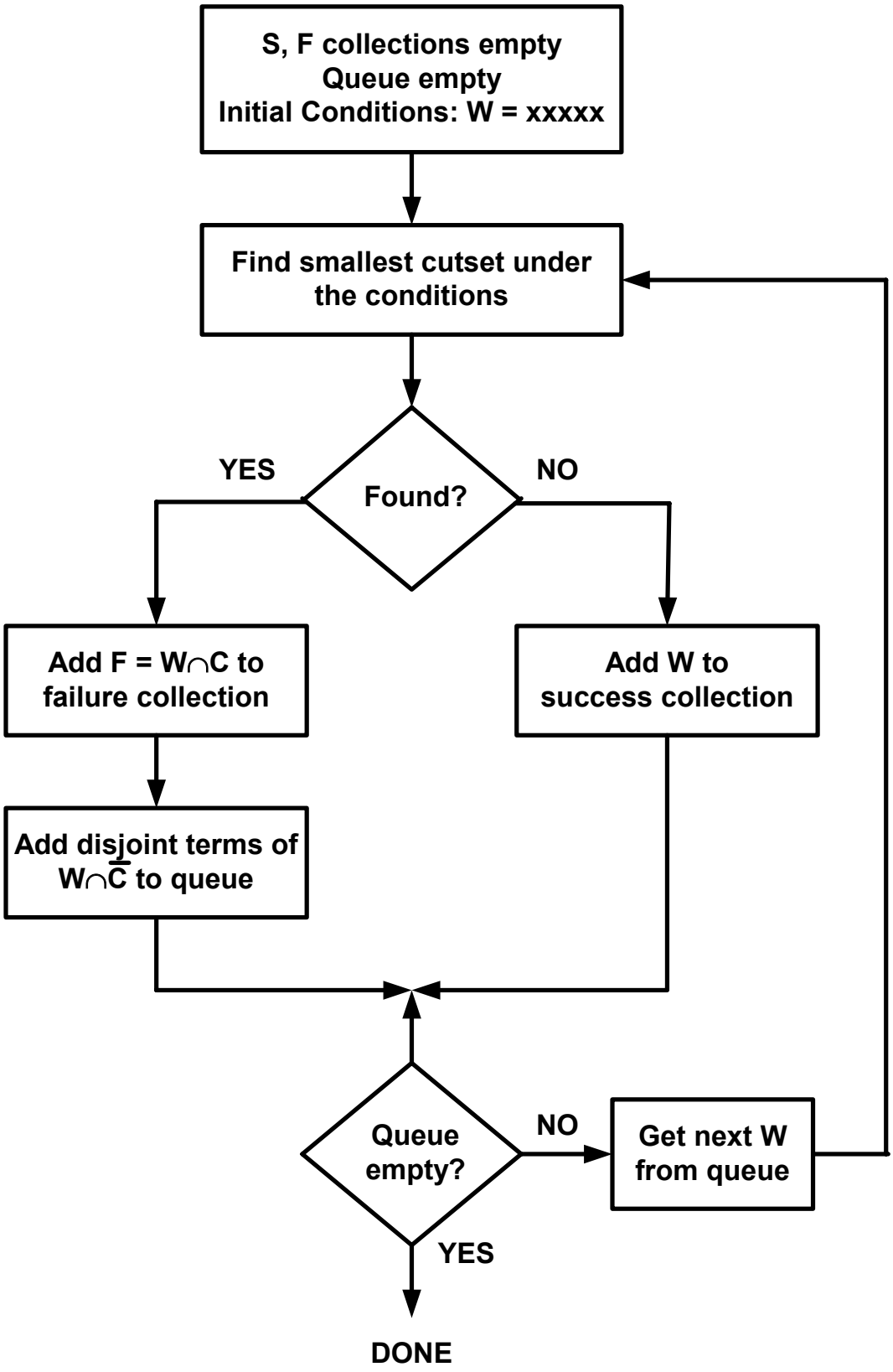


FIGURE 1-3 FLOW DIAGRAM FOR THE ELA2

In Appendix B, an example of the operation of the ELA2 in finding success and failure events for a particular network model is given in detail.

By comparing Figure 1-3 with the flow diagram for the ELA that is given in Figure 1-2, it may be observed that the ELA2 is in some sense a dual to the ELA:

- Whereas the ELA is based on finding s-t paths (sets of links that form an $s \rightarrow t$ connection), the ELA2 is based on finding s-t cutsets (sets of link outages that preclude an $s \rightarrow t$ connection).

- Whereas the ELA forms success events when a path P is found by determining the intersection $S = W \cap P$, the ELA2 forms failure events when a cutset C is found by determining the intersection $F = W \cap C$.

- Whereas the ELA forms next events after a success event has been determined by determining the disjoint terms of the intersection $W \cap \overline{P}$, the ELA2 forms next events after a failure event has been determined by determining the disjoint terms of the intersection $W \cap \overline{C}$.

- Whereas the ELA after classifying an event as a failure ($F = W$) does not process that event further, the ELA2 after classifying an event as a success ($S = W$) does not process that event further.

A difference in the operation of ELA and ELA2 that may be observed is that, while the order in which the disjoint terms of $\overline{P}$ are generated is significant in the ELA, the order in which the disjoint terms of $\overline{C}$ are generated is not significant in the ELA2.

1.2.2 Cutset Search Method

Finding a cutset involves finding either a set of link outages that stops the progression of a path outward from the source or, working from the sink, a set of link outages that stops the backward progression of a path from the sink. Therefore it is expedient to search for a cutset in both directions, in case one is found more quickly in one direction than in another. Another consideration is that the cutset found in one direction may be larger, that is contain fewer link outages, and if so would be preferred because fewer (and larger) next events would be generated. If this second consideration is implemented, then the question of which direction yields a cutset quicker is moot, since cutsets in both directions must be found in order to compare them.

For the postulated event W, let the network description matrix $\boldsymbol{G}$ be altered by setting to 0 the entries that correspond to links that W specifies as being DOWN, and let

U be set of links that are specified in W as being UP. Also, let RF_i denote the set of nodes reached on the ith hop outward from the source node. As suggested in [4], a cutset in the forward direction C_F may be found by forming a set of failable links according the following procedure:

(1) Initialize C_F to include the failable links (if any) in the sth row of the $\boldsymbol{G}$ matrix—those links that are in the sth row but not in U, and therefore were not specified by W as being either UP or DOWN. In order to preclude a path outward from s, these links must fail; therefore, they belong in the cutset. If all of the links in the sth row are failable, then a cutset has been found and the search is stopped.

(2) If one or more of the links in the sth row of $\boldsymbol{G}$ are in U, indicating unfailed connections to other nodes, add those nodes to RF_1 and check whether $t \in RF_1$, that is, whether the node t has been reached in one hop by a link specified by W to be UP. If so, there is no cutset, and the search is stopped.

(3) For the nodes ($\neq t$) in RF_1 , say nodes r_1 and r_2 , add to the cutset the failable links used on the second hop, those in the rows indexed by those nodes (row r_1 and row r_2). If none of the links in these rows is in U, then a cutset has been found and the search is stopped. Otherwise, add the nodes reached by the links in U to the set RF_2 , and check to see if t was reached on the second hop using two links specified in W to be UP; if not, proceed with the cutset search.

(4) Continue for hop i by examining the rows corresponding to the nodes in the set RF_{i-1} , adding failable links in those rows to C_F and stopping with a found cutset if RF_i is empty, or stopping with no cutset found if $t \in RF_i$.

A flow diagram for the forward cutset search is included in Figure 1-4 (left side). The search in the reverse direction for a cutset C_R proceeds similarly, using RR_i to denote the set of nodes reached by links in U on the ith hop backward from the sink:

(1) Initialize C_R to include the failable links (if any) in the tth column of the $\boldsymbol{G}$ matrix—those links that are in the tth column but not in U; if all of the links in the tth column are failable, then a cutset has been found and the search is stopped.

(2) If one or more of the links in the tth column of $\boldsymbol{G}$ are in U, indicating unfailed connections from other nodes, add those nodes to RR_1 and check whether $s \in RR_1$. If so, node t is reachable from node s in one hop using a link specified by W as being UP, there is no cutset, and the search is stopped.

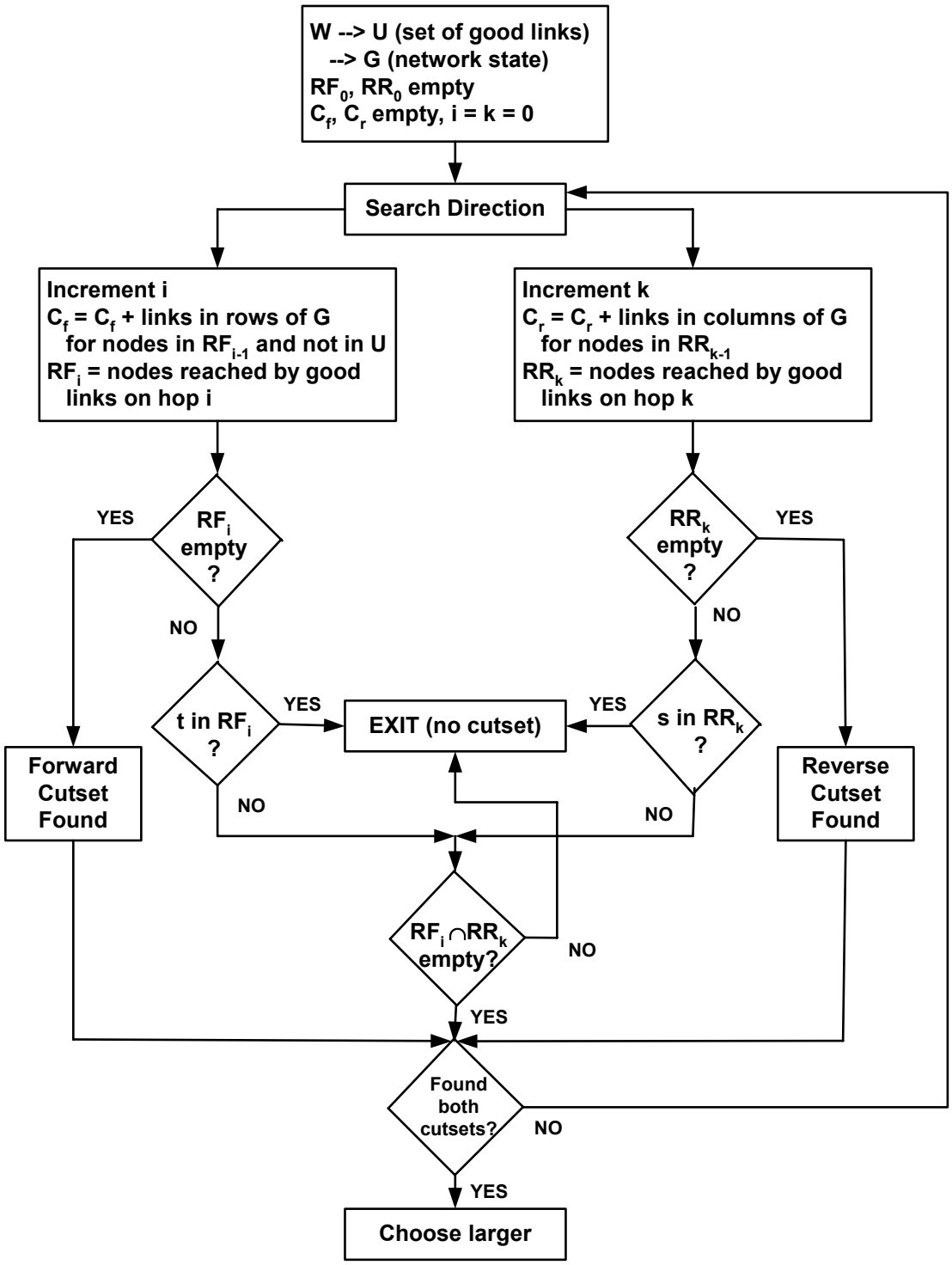


FIGURE 1-4 FLOW DIAGRAM FOR CUTSET SEARCH

(3) For the nodes ($\neq s$) in RR_1 , say nodes c_1 and c_2 , add to the cutset the failable links in the columns indexed by those nodes (column c_1 and column c_2). If none of the links in these columns is in U, then a cutset has been found and the search is stopped. Otherwise, add the nodes reached by the links in U to the set RR_2 .

(4) Continue for hop i by examining the columns corresponding to the nodes in the set RR_{i-1} , adding failable links in those columns to C_R and stopping with a found cutset if RR_i is empty.

A flow diagram for the reverse cutset search is included in Figure 1-4 (right side). Note that if the forward and backward searches are conducted in alternating progressions from the source and sink, respectively, if there is an $s \rightarrow t$ path comprised of links in U this fact can be detected efficiently by an nonempty intersection of RF_k and RR_l for some k and l. This strategy is incorporated in the flow diagram of Figure 1-4, as is the concept of selecting the larger of the forward and reverse cutsets.

In the tests of program implementations discussed later in this report, a comparison is given of the execution times incurred by using two different cutset search strategies: (1) selecting the larger of the forward and reverse cutsets, and (2) selecting the first cutset found.

1.2.3 **An Observation**

At any point during the execution of the ELA or the ELA2, the collection $\mathcal{W}$ consists of events W that completely describe the remaining possibilities for the state of the network links. Moreover, the Ws are disjoint, as are the new (smaller) Ws that are created by removing one of the Ws from $\mathcal{W}$ and processing it—whether by searching for paths or searching for cutsets.

It follows from this observation that there is no reason why a particular W to be examined cannot be arbitrarily examined *either* by the pathfinding method or by the cutset-finding method. For example, the methods employed could be alternated, or perhaps an attempt could be made to minimize the total number of success and failure events by selecting the method based on which one creates fewer new Ws.

In Section 2, tests are made of various algorithms that combine the features of the ELA and the ELA2, in addition to tests of these algorithms in their “pure" form.

1.3 TRUNCATION ISSUES

The work documented by this report was motivated by a desire to improve, if possible, the performance of the ELA in terms of the truncation issues discussed in the following paragraphs.

1.3.1 Bound Convergence

In [1] it was demonstrated that typically the ELA generates success events earlier in the processing than it generates failure events. For this reason the lower bound on the s-t reliability, which is the accumulated probability of success, converges faster than the upper bound, which is the complement of the accumulated probability of failure.

In order to study truncation, in [1] the 4×4 grid network of nodes that is shown in Figure 1-5 was used. The unequal convergence of the lower and upper bounds for the node pair $(s, t) = (2, 15)$ for the case of perfect nodes and links with identical reliabilities ($\beta_0 = 0.6$) is illustrated by the fact that the lower bound converges to within 0.01 of the correct reliability (for example) by the time that the ELA has processed about 4,000 events, while the same accuracy in the upper bound is not reached until about 13,000 events have been processed.

As mentioned previously, a motivation for the ELA2, using cutsets, was the realization that an algorithm based on finding cutsets would tend to accumulate failure probability faster than the original ELA. Figure 1-6 provides a direct comparison of upper and lower bounds for the 3×3 network example worked out in complete detail in Appendices A and B, assuming that $\alpha = 1.0$ for each node and $\beta = 0.5$ for each link. It is clear from this figure that the convergence of the ELA2 bounds (solid lines) is much more even than it is for the ELA bounds (dashed lines).

1.3.2 Estimates Based on the Bounds

It was conjectured in [4] that a reasonable estimate of the s-t reliability could be obtained from the ELA lower bound and the ELA2 upper bound:

$$\hat{\gamma}_{st} = \tfrac{1}{2}\{\mathrm{LB}_{path}(s, t) + \mathrm{UB}_{cutset}(s, t)\}. \tag{1-20}$$

Inspection of Figure 1-7 suggests that averaging the ELA2 bounds might produce a good estimate:

$$\hat{\gamma}_{st} = \tfrac{1}{2}\{\mathrm{LB}_{cutset}(s, t) + \mathrm{UB}_{cutset}(s, t)\}. \tag{1-21}$$

Errors in these two possible estimates are plotted together in Figure 1-8, along with the

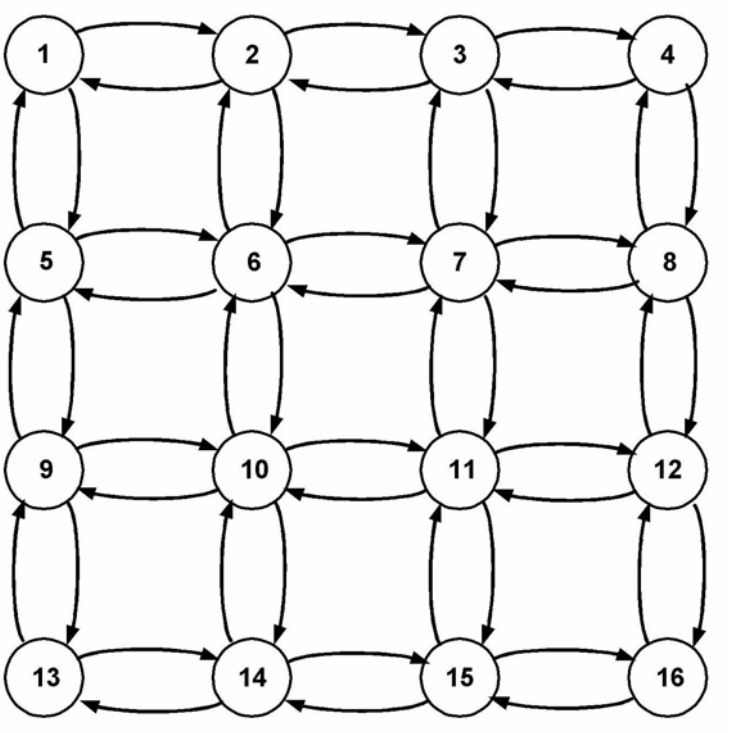


FIGURE 1-5 4×4 EXAMPLE NETWORK

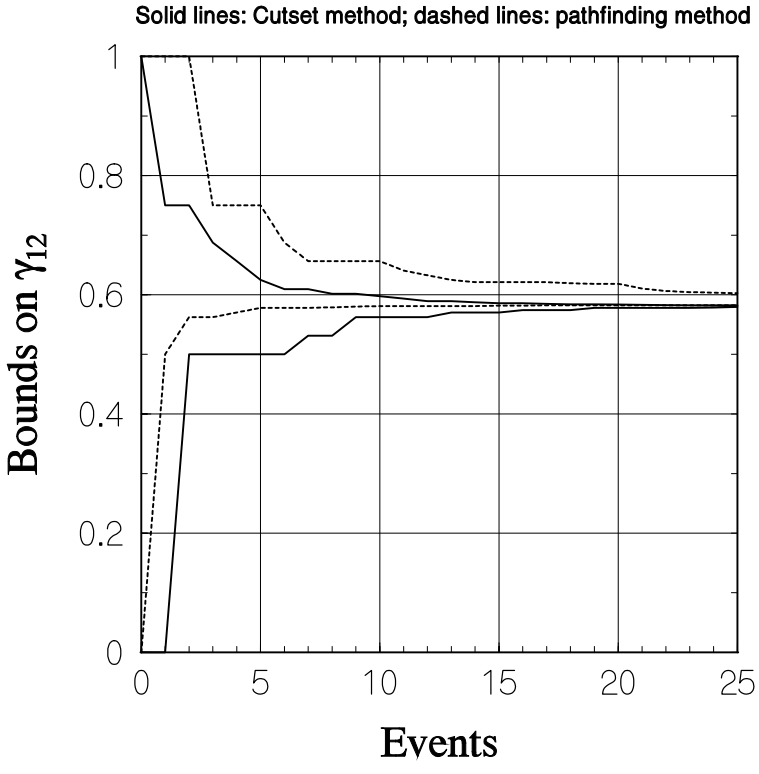


FIGURE 1-6 ELA AND ELA2 BOUNDS

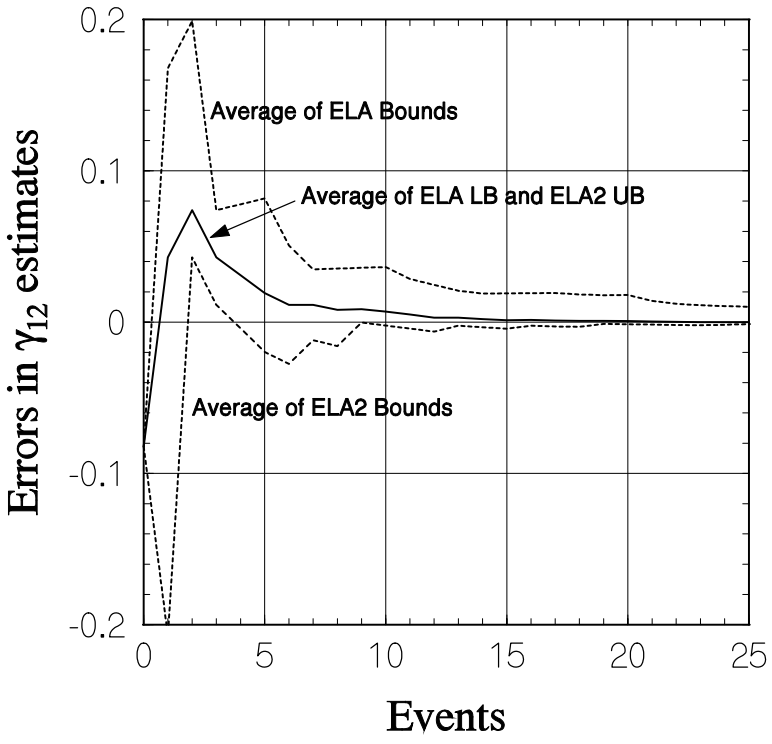


FIGURE 1-8 ERRORS IN ESTIMATES BASED ON BOUNDS

average of the ELA bounds; it appears that both (1-20) and (1-21) are viable candidates for estimates.

1.3.3 Quick Lower Bound

Another concept expressed in [4] was that a lower bound for γ_{st} could calculated quickly by finding a set $\mathcal{P} = \{P_i\}$ of disjoint (s, t) paths (that is, disjoint except for having the source and destination nodes in common), and using the fact that

$$\gamma_{st} \leq \alpha_s \alpha_t \left(1 - \prod_i [1 - \Pr\{P_i\}/\alpha_s \alpha_t] \right). \tag{1-22}$$

However, the number of disjoint paths is limited by the minimum of the number of neighbors of s and the number of neighbors of t, so the effectiveness of this lower bounding technique is very dependent upon the network configuration. For this reason, the technique has not been studied in detail.

2. ALGORITHM DEVELOPMENT

In this section, assessment is made of the relative merits of implementations of the ELA and the ELA2 s-t reliability algorithms and for certain variations on them. The assessment is based on numerical evaluations that are preceded by descriptions of the implementations and by documentation of several network examples that were used for study, in order to introduce the computer file structures that the programs are designed to process.

2.1 NETWORK EXAMPLES

During the development and testing of the s-t reliability algorithms, one or more example networks were used. These example networks are described in the following paragraphs.

2.1.1 3×3 Grid Network Example

This network, shown previously in Figure 1-1, has been used extensively for development because completely worked-out manual solutions have been maintained for this case throughout the project for the case of $(s, t) = (1, 2)$. Appendices A and B of this report contain manual solutions for the ELA and for the ELA2, respectively.

The network has 9 nodes and 24 links; there are fewer than the 72 possible links because the network models an area coverage grid in which each of the radios has highly directional antennas aimed at particular other radios. In order to provide numerical data for algorithm testing, a fictional laydown of a jammer and the 9 nodes was created and processed by the program LINKSNRS described in [5]. That program calculates SNRs for every possible link, based on user-supplied link parameters, and writes them to disk file. Nominal parameters not necessarily realistic in terms of any particular radio system were used, and the disk file was edited by deleting the links not shown in Figure 1-1 to produce the file MESH1000.SNR.[1]

A list of the links and link reliabilities for this example is given in Table 2-1 at the end of subsection 2.1.4, and was calculated assuming

$$\beta_{ij} = \mathrm{P}_G\left(\frac{\mathrm{SNR}_{ij}-\mu}{\sigma_L}\right) = \mathrm{P}_G\left(\frac{\mathrm{Margin}_{ij}}{\sigma_L}\right), \tag{2-1}$$

[1]The format of the disk files is discussed in detail below.

where $P_G(\cdot)$ is the Gaussian cumulative distribution function, μ is the system's SNR threshold criterion in dB, and σ_L is a standard deviation for the SNR in dB. As suggested in [6], σ_L is taken to be 10 dB, while the value of μ is system-dependent; for the data in Table 2-1 and unless otherwise noted, a μ value of zero dB is assumed.[2] For this report, network examples were selected in which generally the links all have a positive margin, since the focus of the study is on the operation of network analysis algorithms when a large number of links are viable. Since a link is DOWN for $SNR_{ij} < \mu$, the analysis programs treat links with $\beta_{ij} < 0.5$ as being absent; this treatment has the effect of defining β_{ij} to have zero value for $SNR_{ij} < \mu$, as far as s-t reliability is concerned. However, the actual values of link reliability are preserved so that they can be retrieved individually if desired.

2.1.2 **15-Node Network Example**

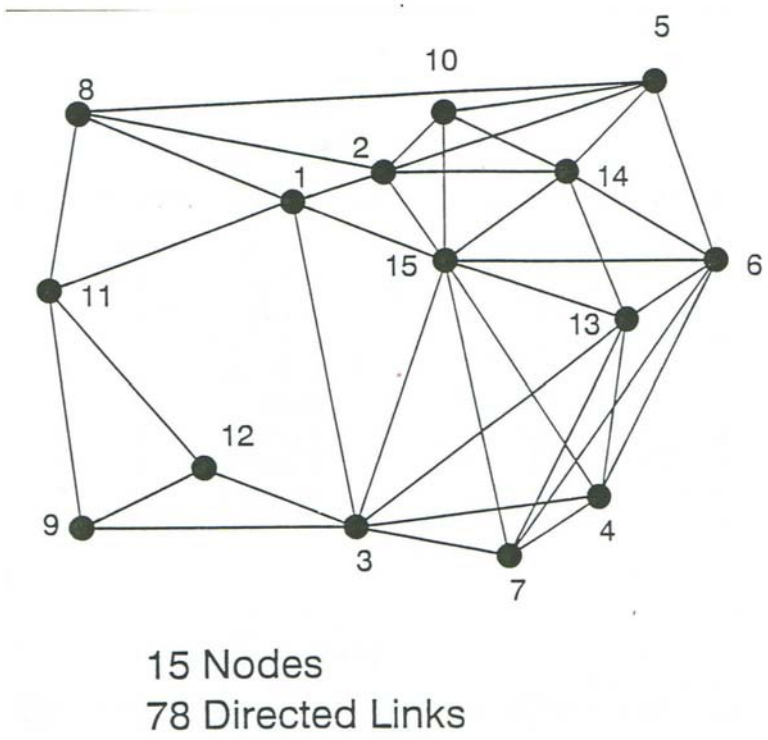
The 15-node network shown in Figure 2-1(a) is based on a network representing a tactical radio deployment that was studied extensively in [8] using variations on the Page-Perry algorithm ([9]-[11]) to calculate s-t reliability or bounds on s-t reliability. Rather than being a fully connected network (with 210 possible links), for algorithm testing purposes many of the possible links have been deleted, in effect simulating a network with omnidirectional antennas and terrain obstructions. The associated disk file is T920811.SNR.

The particular 15-node network example used most frequently in the algorithm testing for this report is the case shown in Figure 2-1(b), which differs from that of Figure 2-1(a) in that several links have dropped out due to jamming effects; note that some of the nodes are connected in only one direction. A list of the links and their reliabilities for the example case is given in Table 2-2. The particular s-t pair studied was (8, 13).

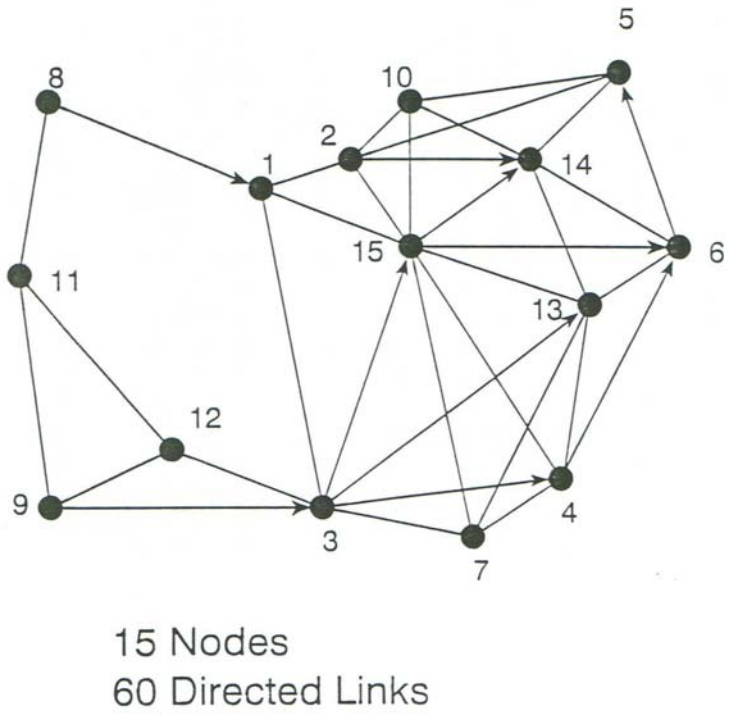
2.1.3 **34-Node Network Example**

The 34-node network example shown in Figure 2-2 represents a realistic deployment of an area coverage network; 32 of the nodes correspond to net control stations and two of the nodes are relays. There are 128 directional links, which are listed in Table 2-3 with their reliabilities for a particular case (file BIGONE.SNR).

[2]Later in this section, the value of the threshold μ will be varied for parametric studies.



(a) Network configuration based on terrain.



(b) Viable links for a particular jamming situation.

FIGURE 2-1 15-NODE NETWORK EXAMPLE

TABLE 2-1

LINK RELIABILITIES FOR 3 × 3 EXAMPLE NETWORK

i	j	β_{ij}	i	j	β_{ij}	i	j	β_{ij}
1	2	0.785157	4	5	0.755804	6	9	0.839431
1	4	0.851968	4	7	0.862287	7	4	0.851968
2	1	0.862287	5	2	0.785157	7	8	0.785157
2	3	0.839431	5	4	0.851968	8	5	0.755804
2	5	0.755804	5	6	0.536532	8	7	0.862287
3	2	0.785157	5	8	0.785157	8	9	0.839431
3	6	0.536532	6	3	0.839431	9	6	0.536532
4	1	0.862287	6	5	0.755804	9	8	0.785157

TABLE 2-2

LINK RELIABILITIES FOR 15-NODE EXAMPLE NETWORK

i	j	β_{ij}	i	j	β_{ij}	i	j	β_{ij}
1	2	0.835958	5	14	0.630889	12	3	0.887471
1	3	0.752632	6	5	0.511271	12	9	0.799649
1	15	0.865923	6	13	0.525386	12	11	0.713101
2	1	0.555662	6	14	0.883352	13	4	0.765767
2	5	0.717830	7	3	0.778150	13	6	0.811777
2	10	0.998010	7	4	0.982821	13	7	0.716778
2	14	0.786356	7	13	0.762751	13	14	0.902274
2	15	0.917022	7	15	0.661659	13	15	0.601432
3	1	0.737496	8	1	0.670613	14	5	0.639387
3	4	0.732484	8	11	0.881726	14	6	0.920422
3	7	0.822769	9	3	0.721325	14	10	0.520149
3	12	0.632344	9	11	0.795816	14	13	0.862100
3	13	0.511243	9	12	0.961764	15	1	0.593906
3	15	0.741865	10	2	0.971308	15	2	0.912072
4	6	0.529203	10	5	0.750600	15	4	0.687622
4	7	0.877594	10	14	0.812209	15	6	0.568354
4	13	0.819577	10	15	0.901760	15	7	0.675236
4	15	0.690958	11	8	0.921801	15	10	0.938239
5	2	0.669306	11	9	0.787474	15	13	0.872102
5	10	0.697838	11	12	0.678915	15	14	0.759407

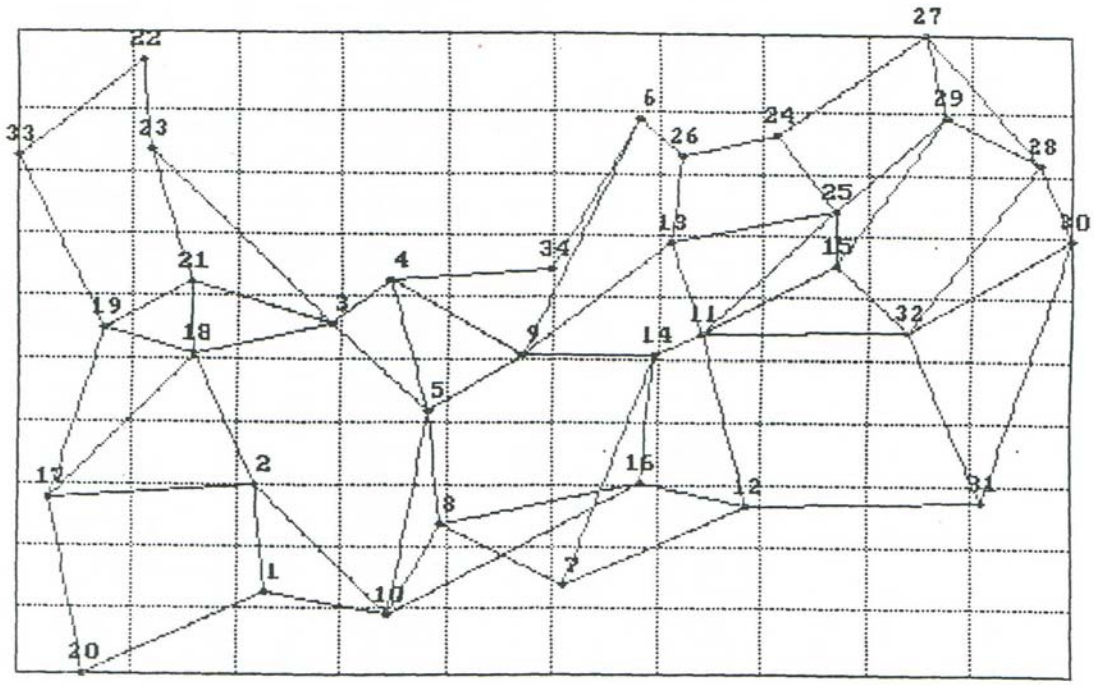


FIGURE 2-2 34-NODE NETWORK EXAMPLE

TABLE 2-3

LINK RELIABILITIES FOR 34-NODE EXAMPLE NETWORK

i	j	β_{ij}	i	j	β_{ij}	i	j	β_{ij}
1	2	0.999395	11	14	0.996267	22	23	0.999726
1	10	0.997848	11	15	0.987425	22	33	0.998290
1	20	0.666368	11	25	0.982805	23	3	0.990083
2	1	0.999407	11	32	0.967023	23	21	0.998718
2	10	0.995056	12	7	0.521169	23	22	0.999735
2	17	0.649337	12	11	0.991438	24	25	0.996583
2	18	0.998512	12	16	0.979619	24	26	0.982150
3	4	0.999600	12	31	0.952729	24	27	0.977501
3	5	0.997623	13	9	0.960288	25	11	0.988881
3	18	0.730653	13	11	0.998150	25	13	0.516606
3	21	0.984823	13	25	0.982888	25	15	0.999453
3	23	0.992793	13	26	0.998776	25	24	0.997350
4	3	0.996862	14	7	0.988226	25	29	0.983336
4	5	0.997941	14	9	0.658081	26	6	0.996076
4	9	0.995204	14	11	0.999518	26	13	0.998813
4	34	0.993353	14	16	0.997343	26	24	0.996244
5	3	0.998071	15	11	0.957977	27	24	0.985568
5	4	0.998082	15	25	0.999455	27	28	0.965768
5	8	0.998907	15	29	0.975061	27	29	0.996825
5	9	0.997993	15	32	0.994441	28	27	0.917088
5	10	0.994961	16	8	0.549694	28	29	0.954365
6	9	0.986123	16	10	0.466321	28	30	0.992770
6	26	0.999496	16	12	0.995581	28	32	0.969944
6	34	0.994154	16	14	0.997052	29	15	0.982958
7	8	0.997035	17	2	0.993678	29	25	0.988907
7	12	0.983109	17	18	0.995396	29	27	0.997372
7	14	0.983774	17	19	0.997842	29	28	0.985658
8	5	0.998888	17	20	0.998125	30	28	0.994272
8	7	0.996155	18	2	0.998363	30	31	0.951681
8	10	0.999109	18	3	0.997163	30	32	0.892092
8	16	0.986373	18	17	0.996472	31	12	0.901892
9	4	0.978642	18	19	0.995453	31	30	0.915909
9	5	0.988973	18	21	0.999848	31	32	0.980438
9	6	0.981803	19	17	0.998079	32	11	0.432668
9	13	0.988674	19	18	0.999323	32	15	0.995832
9	14	0.994613	19	21	0.999312	32	28	0.948248
10	1	0.988527	19	33	0.997529	32	30	0.944548
10	2	0.995988	20	1	0.994567	32	31	0.977857
10	5	0.994060	20	17	0.998133	33	19	0.997160
10	8	0.998937	21	3	0.996806	33	22	0.997905
10	16	0.973888	21	18	0.999847	34	4	0.636755
11	12	0.991010	21	19	0.995378	34	6	0.992709
11	13	0.998325	21	23	0.998817			

A goal of the project is to discover ways to make survivability analysis for networks of this size convenient on small computers. A measure of network survivability is the connectivity, or average s-t reliability, given by

$$\overline{\gamma} = \frac{1}{N(N-1)} \sum_{\substack{i=1 \\ i \neq j}}^{N} \sum_{j=1}^{N} \gamma_{ij} \,. \tag{2-2}$$

In [1] and [7], the connectivity of the 34-node network model was estimated using

$$\hat{\overline{\gamma}} = \frac{1}{M} \sum_{m} \hat{\gamma}_{i_m j_m} \,, \tag{2-3}$$

where instead of averaging over all node pairs as in (2-2), the averaging was over a set of $M = 37$ node pairs $\{(i_m,\ j_m);\ m = 1,\ 2,\ \ldots,\ M\}$ selected as representative of the distribution of node-pair hop distances. Also in (2-3), $\hat{\gamma}_{i_m j_m}$ denotes an estimate of the s-t reliability for the mth pair; the estimate was taken to be the partial sum of success probabilities (a lower bound); tests using selected pairs of the 4×4 grid network showed that the lower bound was reasonably tight when 70% of the total number of ELA events had been evaluated, giving less than a 0.3 difference between the upper and lower bounds for the special case of all the links having the reliability $\beta_0 = 0.5$. Thus the ELA was truncated for a pair when one of the following conditions was satisfied:

(a) a lower limit of 100 success and failure events have been enumerated, AND at least 70% of the elementary events have been accounted for; or

(b) an upper limit of 10,000 success and failure events have been enumerated, or

(c) the algorithm has run to completion.

The lower limit of 100 events was instituted because some node pairs (such as those with high probability because they are separated by one hop) may reach a lower bound value greater than 0.7 very quickly, forcing $\mathrm{UB} - \mathrm{LB} < 0.3$ regardless of the number of failures; it was deemed advisable therefore to require a nominal minimum of 100 events unless, of course, the algorithm runs to completion for fewer than 100 events.

The upper limit of 10,000 events was instituted because of the desire to keep the resulting file size small enough (< 1.44 Mb) to store conveniently on a floppy disk for backup storage and/or portability of the data. Although a number of 10,000-event files could be employed for a given node pair, it also was desired to be able to maintain the event files on the computer's hard disk during any calculations that use them, and so a

10,000-event limit was selected as a reasonable compromise to allow a number of node-pair event files to reside in the available disk space simultaneously.

Using these truncation procedures, the success and failure events were enumerated for a number of node pairs, of varying hop distances and orientations, for the 34-node model network of Figure 2-2. The relative numbers of pairs selected with given hop distances (h_{min}) is roughly equivalent to the distribution of hop distances, which was found to be as shown in Table 2-4 for the example network. Data pertaining to the pairs so examined are listed in Table 2-5.

The quantity "% elem." given in the seventh column of Table 2-5 is the percentage of the 2^M elementary events, where M is the number of links, that are included in the possible network conditions described by the success and failure events in the (partial) success and failure collections. This percentage was measured by calculating the sum of the probabilities of the success and failure events for the special case of all the links having a reliability value of $\beta = 0.5$, so that one elementary event has the probability $(0.5)^M = 2^{-M} = 1/2^M$.

TABLE 2-4 DISTRIBUTION OF HOP DISTANCES

Example network of Figure 2-2

hop distance	number of pairs	percentage
1	120	12.1
2	202	20.4
3	214	21.6
4	176	17.7
5	126	12.7
6	88	8.9
7	52	5.2
8	14	1.4
	$992 = 32 \times 31$	

TABLE 2-5
NODE PAIRS ENUMERATED USING THE TRUNCATION RULES

Network model: 34-node network of Figure 2-2
9-hop limit on paths
"% elem." ≡ percentage of elementary events analyzed

	(s, t)	h_{min}	#events	#success	#failure	% elem.	truncation rule
1	(1, 2)	1	100	80	20	93	≥ 100 events
2	(5, 9)	1	102	100	2	77	≥ 100 events
3	(17, 19)	1	101	77	24	88	≥ 100 events
4	(26, 13)	1	101	89	12	89	≥ 100 events
5	(14, 16)	1	101	95	6	80	≥ 100 events
6	(31, 32)	1	100	87	13	92	≥ 100 events
7	(14, 15)	2	906	844	62	70	≥ 70% elem. ev.
8	(22, 21)	2	100	75	25	96	≥ 100 events
9	(17, 10)	2	187	175	12	71	≥ 70% elem. ev.
10	(25, 27)	2	203	177	26	70	≥ 70% elem. ev.
11	(18, 4)	2	782	678	104	70	≥ 70% elem. ev.
12	(13, 14)	2	195	183	12	70	≥ 70% elem. ev.
13	(5, 16)	2	357	344	13	70	≥ 70% elem. ev.
14	(6, 7)	3	2,857	2,505	352	70	≥ 70% elem. ev.
15	(1, 3)	3	201	193	8	70	≥ 70% elem. ev.
16	(24, 30)	3	982	858	124	70	≥ 70% elem. ev.
17	(3, 6)	3	1,950	1,243	707	70	≥ 70% elem. ev.
18	(16, 25)	3	6,848	6,291	557	70	≥ 70% elem. ev.
19	(9, 32)	3	10,000	8,968	1,032	69	≤ 10,000 events
20	(13, 8)	3	5,460	4,853	607	70	≥ 70% elem. ev.
21	(3, 11)	4	10,000	8,724	1,276	69	≤ 10,000 events
22	(2, 22)	4	2,196	1,418	776	70	≥ 70% elem.ev.
23	(12, 27)	4	10,000	8,065	1,935	68	≤ 10,000 events
24	(8, 30)	4	10,000	8,686	1,314	66	≤ 10,000 events
25	(18, 14)	4	10,000	9,078	922	65	≤ 10,000 events
26	(6, 15)	4	735	697	38	70	≥ 70% elem. ev.
27	(18, 25)	5	10,000	9,731	269	44	≤ 10,000 events
28	(12, 19)	5	10,000	8,882	1,118	57	≤ 10,000 events
29	(8, 29)	5	10,000	9,515	485	55	≤ 10,000 events
30	(13, 17)	5	10,000	9,223	777	51	≤ 10,000 events
31	(21, 32)	6	10,000	9,637	363	46	≤ 10,000 events
32	(30, 3)	6	10,000	9,002	998	47	≤ 10,000 events
33	(25, 20)	6	10,000	8,478	1,522	31	≤ 10,000 events
34	(19, 29)	7	10,000	9,455	545	36	≤ 10,000 events
35	(27, 1)	7	10,000	8,838	1,162	40	≤ 10,000 events
36	(20, 27)	8	1,366	1,202	104	70	≥ 70% elem. ev.
37	(30, 22)	8	6,129	3,427	2,702	70	≥ 70% elem. ev.
			184,789				

2.2 ALGORITHM IMPLEMENTATION

In this subsection the computer program implementations of various s-t reliability calculation algorithms are described. All of the programs are written in TurboPascal and were compiled in the Borland TurboPascal 6.0 environment.

2.2.1 **Common Program Structure**

Since all of the programs have the same purpose, there is a common program structure, which can be understood from the flow diagram in Figure 2-3. In the following paragraphs, the steps in the common program flow are explained.

Read SNR File. The programs are set up to read a file describing the network in terms of the SNRs at the receivers on given directed links for a particular scenario. A sample SNR file is shown in Figure 2-4. It was generated using the utility program LINKSNRS that is documented in [5] and has the format shown.

The first line of the file gives the names of the files describing the network node positions (*.XYN) and the jammer positions, orientations, and powers (*.JAM). LINKSNRS uses the information in these files, plus link budget information supplied by the user, to calculate jammed SNRs and to write them to a file with a .SNR extension. A sample of the LINKSNRS summary of parameters is given in Figure 2-5.

The second line of the file has the number 1 followed by the number of nodes and the number of jammer cases (one of which may be selected); the sample SNR file in Figure 2-4 is the source of the link reliability data in Table 2-3 in which there are 34 nodes, and jammer case 4 was used.

The third and subsequent lines of an SNR file list the links in terms of the numbers assigned to the transmitter and receiver nodes, in that order, and then, on the same line, the SNRs at the receiver on the link for as many jammer cases as there were specified on the second line of the file. Note that the fact that N nodes have been declared does not imply anything about the number of links listed in the file; not all ordered node pairs have to appear in the file. However, LINKSNRS generates numbers for all node pairs, and lines in the file can be deleted, as they were in Figure 2-4, to indicate the absence of a link due the use of directional antennas, terrain obstructions, etc.

Get parameters. The second step in the common program flow is to obtain from the user certain parameters needed to perform the s-t reliability calculation. These include μ and σ_L for calculating link reliabilities according to (2-1); numbers indexing the source

and destination nodes ($i.\ e.$, s and t); the index of the jamming case; and for some of the programs, a single value of node reliability (α).

Calculate link reliabilities. The third step in the common program flow is to calculate link reliabilities. As these calculations proceed, the programs assign an index number to each of the links, in the order illustrated in the rows of the matrix shown in Figure 1-1, and the node and link information is loaded into whatever data structure is being used for network representation.

Calculate s-t reliability. The fourth step in the common program flow is to calculate the reliability of the connection for the given (s, t) node pair, and/or bounds on that reliability.

Display results. The last step in the common program flow is to display the results of the calculations. The user may also be permitted to run another node pair or another jamming case for the same node pair.

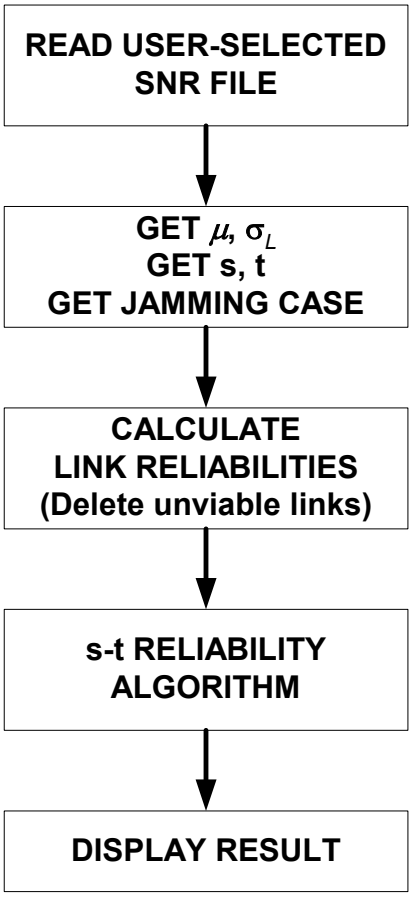


FIGURE 2-3 COMMON STRUCTURE OF COMPUTER PROGRAMS

Inputs=		XY34.XYN		DUMMY	EASTLOW.JAM
1	34	4			
1	2	41.205809	39.326995	37.746430	32.380179
1	10	37.982139	35.851854	34.135723	28.558982
1	20	21.106659	14.265242	11.273839	4.299316
2	1	41.214147	39.354102	37.784152	32.435107
2	10	35.226968	33.096683	31.380552	25.803811
2	17	20.642225	13.801321	10.809984	3.835514
2	18	38.287219	36.508215	34.984932	29.712867
3	4	43.253863	40.995421	39.214668	33.543256
3	5	38.133867	35.792232	33.971031	28.242092
3	18	22.993220	16.121252	13.125896	6.148187
3	21	34.997649	30.716332	28.203409	21.657182
3	23	32.829746	31.134394	29.660750	24.473033
4	3	41.229434	36.578179	33.973496	27.337984
4	5	38.591337	36.249702	34.428501	28.699562
4	9	36.443387	33.791294	31.828802	25.908636
4	34	35.525009	32.757811	30.746525	24.763610
5	3	38.255587	36.157167	34.457556	28.905582
5	4	38.634129	36.375687	34.594934	28.923522
5	8	40.499024	38.177078	36.365345	30.649757
5	9	39.315588	36.663496	34.701004	28.780838
5	10	35.160861	33.030575	31.314445	25.737704
⋮	⋮	⋮	⋮	⋮	⋮
31	12	28.305762	22.580159	19.757257	12.923524
31	30	28.841422	23.355706	20.576083	13.780173
31	32	34.179182	29.760055	27.211803	20.630759
32	11	15.242859	8.294898	5.289844	-1.695655
32	15	39.365863	35.319886	32.870669	26.388675
32	28	31.026310	25.772162	23.037094	16.280873
32	30	31.002452	25.516736	22.737113	15.941203
32	31	33.938261	29.333914	26.740334	20.115430
33	19	35.781191	34.175408	32.756642	27.665288
33	22	36.921800	35.256329	33.800790	28.644496
34	4	20.385340	13.479127	10.479376	3.498135
34	6	35.779267	32.700316	30.565665	24.431009

FIGURE 2-4 SAMPLE SNR FILE CONTENTS

```
PARAMETERS FOR CALCULATION                    DEFAULT
Total Rcvr and Env Noise (dBm):               -106.4
Power Delivered to Antenna (dBm):               37.0
Transmitter Antenna Gain (dBi):                 20.0
Transmitter Insertion Loss (dB):                 5.4
Receiver Mainbeam Gain (dBi):                   20.0
Receiver Mainbeam Limit (degr):                  9.0
Receiver Sidelobe Gain (dBi):                    4.0
Receiver Sidelobe Limit (degr):                 28.0
Receiver Backlobe Gain (dBi):                   -3.0
Receiver Insertion Loss (dB):                    5.4
Other Insertion Loss (dB):                       2.0
Power Amplifier Gain (dB):                       0.0
Signal Transmission Power (W):                   5.0
Bandwidth Reduction Factor (dB):                10.7
Jammer Polarization Loss (dB):                   3.0
Transmission Frequency (MHz):                 1500.0
AL1:                                           11.05
AL2:                                             0.0
AL3:                                            0.07
EPL  = AL1 + AL2 * log d + AL3 * d
  Hit F10 to Accept
  Up/Down Arrow to Move, Enter to Change
```

FIGURE 2-5 SAMPLE LINK BUDGET PARAMETER SUMMARY

2.2.2 **Programs Based on Partitioning**

In the following paragraphs, brief descriptions are given for the computer programs implementing the equivalent-links algorithm and variations on it.

2.2.2.1 Program with pathfinding (ELA)

The program EQLNKTST listed in Appendix D.1 implements the ELA and produces upper and lower bounds for the *s*-*t* reliability of a given node pair for a specified jamming case. EQLNKTST, as listed, contains procedures and functions to facilitate the user interface, and makes use of other procedures and functions residing in the TurboPascal units EQLINKS and NETSET, the latter of which is listed separately in Appendix D.5 because it is also accessed by other programs.

Prior to the reliability calculation, EQLNKTST calls the procedure *GettNet* (a procedure included in the unit NETSET), which creates a network description from the

information supplied by the user through keyboard entry and SNR values for the network links in the *.SNR file specified by the user. The network description is in the form of a special data structure including the following components:

```
Network  = RECORD
  Source   : Integer;
  Sink     : Integer;
  NodeNum  : Integer;
  EdgeNum  : Integer;
  MaxHops  : Integer;
  GraphMat : AdjcMatPtr;
  Beta     : EdgeVectPtr;
  Alpha    : NodeVectPtr;
  PathList : EdgeListPtr;
  I_Index  : EdgeListPtr;
  J_Index  : EdgeListPtr;
  UpEdges  : String;
END; (* Network - basic network structure *)
```
(2-4a)

where

```
NodeList = ARRAY [1..NODEMAX] OF Byte;
NodeVect = ARRAY [1..NODEMAX] OF Real;
EdgeList = ARRAY [1..EDGEMAX] OF Byte;
EdgeVect = ARRAY [1..EDGEMAX] OF Real;
AdjcMat  = ARRAY [1..NODEMAX] OF NodeList;
NodeListPtr = ^NodeList;
NodeVectPtr = ^NodeVect;
EdgeListPtr = ^EdgeList;
EdgeVectPtr = ^EdgeVect;
AdjcMatPtr  = ^AdjcMat;
```
(2-4b)

In these Pascal statements, it is assumed that maximum numbers of nodes and edges (links) have been defined as constants; the program necessarily oversizes the arrays because they cannot be changed dynamically in Pascal. However, the arrays actually making up the network data structure are “pointer" variables (those given names ending in “...Ptr"), which do not take up computer memory until the variable is actually used, allowing for dynamic memory allocation of a sort if necessary.

In order to understand how the program operates, it is only necessary to realize that for a particular case, in the mathematical notation used in Section 1, there are N nodes and M links. So the network data structure described in the statements above consists essentially of the parameters s ($= Source$), t ($= Sink$), N ($= NodeNum$), M ($= EdgeNum$), a user-supplied limit on the number of hops in a path ($= MaxHops$), an $N \times N$ adjacency matrix ($= GraphMat$), a $1 \times M$ array of link reliabilities ($= Beta$), and a $1 \times N$ array of node reliabilities ($= Alpha$).

Also associated with the network description in the Pascal statements are a $1 \times M$ array (*PathList*) reserved for listing (in order) the links used by an $s \rightarrow t$ path, a string variable (*UpEdges*) used to indicate which links are used by a path irrespective of path order, and two $1 \times M$ arrays giving the starting nodes (*I_Index*) and stopping nodes (*J_Index*) for each link.

The procedure *GettNet* examines the SNRs for all $N(N-1)$ possible links and skips possible links that are considered to be unviable—those with SNRs below threshold. The viable links ($i \rightarrow j$) are assigned numbers as they are entered into the (i, j) positions of the adjacency matrix, as illustrated in Figure 1-1. In effect, the adjacency matrix is a table that can be quickly consulted to determine whether two nodes are directly connected, and if so, by what link. After assigning a number k to a viable link connecting node i to node j, for convenience as a cross-reference the program records the facts that *I_Index*(k) $= i$ and *J_Index*(k) $= j$.

Having “loaded" the network description information, the calling program EQLNK-TST then activates the procedure *ELReliab*, which returns upper and lower bounds on the s-t reliability. This procedure is part of the EQLINKS unit, utilizes procedures and functions in EQLINKS and in NETSET, and may be diagrammed as shown in Figure 2-6. That figure is intended to be self-explanatory, but to aid in comprehending it the following remarks are made:

Two sequentially accessed hard-disk files are used for queues[3], one opened for input—having been filled with network events to be tested for success or failure—and one opened for output, to be filled with new events that result when the ELA finds that an event it has tested gives rise to a success. Two files are used—even though conceptually only one queue is required—because it is the most practical way to implement the queue. For example, taking an event from the queue involves *removing* it from the queue; while it is a simple matter to read an event from a file, there is no practical way then to delete that single event from the file. The programs simply process all of the events contained in a “read" or input file, the results being put into a “write" or output file; then the input file is erased (overwritten, actually) and the former output file becomes the input file. Thus when all of the events in one file have been tested, the program “swaps" files and continues testing events; this process is terminated when there are no more new events

[3]Hard-disk files are used for queues because the amount of memory needed can be a megabyte or more; the I/O to access these files affects the speed of the program considerably.

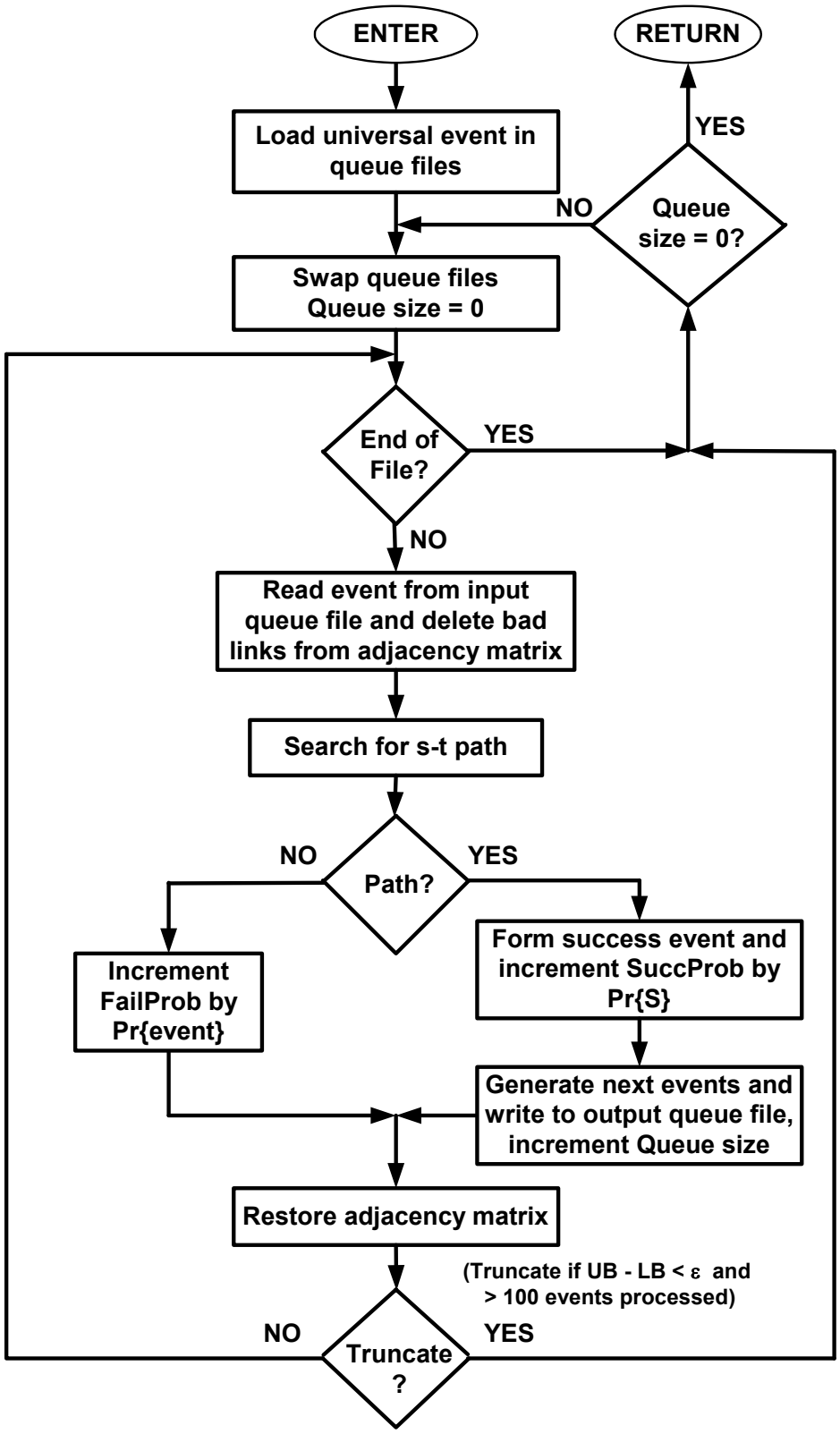


FIGURE 2-6 FLOW FOR THE ELA IMPLEMENTATION

added to the output file (which occurs when all the events in the input file turn out to be failure events).

The integer variable *QueueSize* referred to in Figure 2-6 is the number of new events that have been written to the output file. After all the events in the input file have been processed, the program knows whether it is done by checking on the value of *QueueSize*: if *QueueSize* > 0, then the program is not done.

An event is determined to be a success if a path can be found from s to t, given the conditions of the event being tested. One path search technique utilized in the implementation of the ELA is a one-way search, which in effect simulates the transmission of messages from the source node to adjacent nodes on the first hop, further transmission on the second hop to nodes adjacent to those already reached, etc., until either the sink node is reached or it is determined that the sink cannot be reached or has not been reached in the specified maximum number of hops.

In Appendix E it is shown that the overall speed of the ELA is affected very much by the efficiency of the path search technique. For that reason, four different versions of pathfinding were compared, the possible combinations of the following two properties: (1) One- or two-way path search; a two-way search works both forward from s and backward from t. (2) Path search with anti-pingpong logic or with anti-return logic. By “anti-pingpong" is meant the prevention of the search from progressing to a node from which it emanated on the previous hop. By “anti-return" is meant the prevention of the search from progressing to *any* node previously visited. These measures in different degrees eliminate “loops" in the path search that use up computer time but do not affect the outcome of the search. On the average, the best performance in terms of execution time results from using a two-way search and anti-return logic.

Once a sequence of links forming a path has been found, the string *UpEdges* encodes those links and is used to find the intersection of the path with the event giving rise to the success ($S = W \cap P$ in the notation of Section 1.1). For example, if the event being tested is

$$W = \texttt{021111221111111111121121}, \tag{2-5a}$$

which indicates[4] that the link numbered $N+1$ is specified in the event to be DOWN and the links $N+2$, $N+7$, $N+8$, $N+21$, and $N+24$ are specified to be UP. Further, suppose that the encoded path is

[4]For programming convenience, the states -1, 0, and 1 spoken of in Section 1.1 are shifted to the (string) values 0, 1, and 2.

$$UpEdges = 121112111111111121111111, \tag{2-5b}$$

which indicates that the path consists of the successes of links $N+2$, $N+6$, and $N+18$. Then the success event is determined by the procedure *Success* in EQLINKS to be

$$S = 021112221111111121121121. \tag{2-5c}$$

The order of the sequence is preserved in the array *PathList* and used to generate new events; it was shown in [1] that preserving this order generates fewer events to be tested. Note that in the example of (2-5), the successes of links $N+6$ and $N+18$ are new conditions, while that of link $N+2$ is not; therefore, although there are three links in the path, only two new events are generated.

The process is terminated early (truncated) when the lower bound, which is the sum of the probabilities for the success events found so far, is within ϵ of the upper bound, which is one minus the sum of the probabilities for the failure events found so far, provided that at least 100 events have been tested. The criterion ϵ is a constant that is "hard-wired" into the program code, nominally with the value $\epsilon = 0.01$.

2.2.2.2 Program with cutsets (ELA2)

The program EL2ONLY listed in Appendix D.2 implements the ELA2 using functions and procedures in the unit CUTONLY and in the unit NETSET. In a manner analogous to the implementation of the ELA, the program EL2ONLY acts as the user interface and calling program, while the unit CUTONLY provides ELA2-specific routines and the unit NETSET provides routines that are common to several programs. The data structure used for representing the network and the method for loading it are identical with that for the implementation of the ELA, as described in Section 2.2.2.1.

Having loaded the network description information, the calling program EL2ONLY then activates the procedure *ELReliab*, which returns upper and lower bounds on the s-t reliability. Though it has the same name as the procedure used to implement the ELA, this procedure, which is part of the unit CUTONLY, in processing queued events operates in a completely complementary fashion, as diagrammed in Figure 2-7.

In place of the search for a path, the implementation of the ELA2 searches for a cutset. The search is done in two directions, as discussed in Section 1.2.2. An event is determined to be a failure if a cutset of hypothethically failed links can be found that precludes a path from s to t, given the conditions of the event being tested. If a cutset

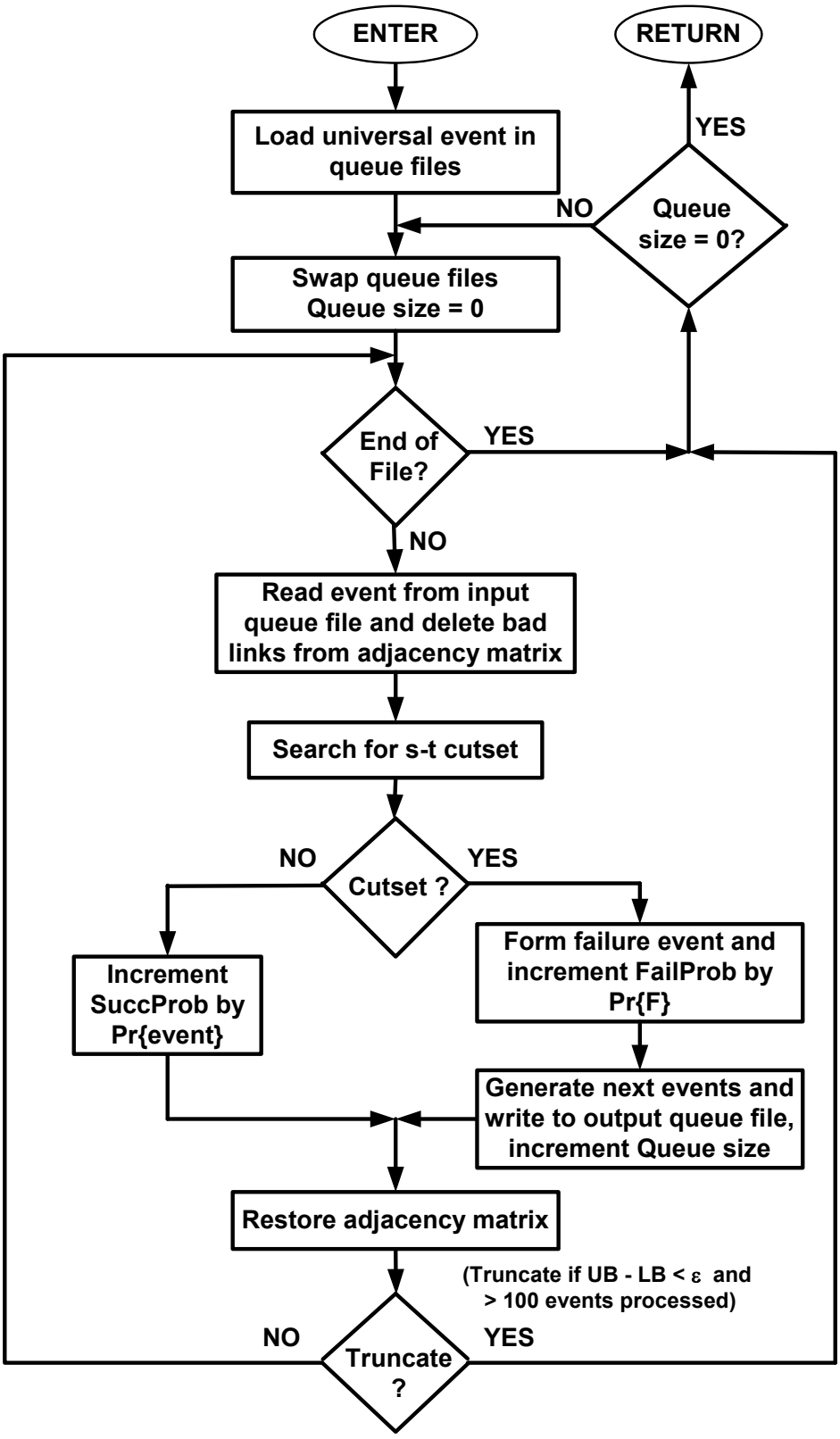


FIGURE 2-7 FLOW FOR THE ELA2 IMPLEMENTATION

cannot be found, it is because there is an $s \rightarrow t$ path consisting entirely of links that are specified to be UP.

Once a cutset of link failures has been found, the string *UpEdges* encodes those links (positively) and the array *PathList* is used as a means to list the links whose failures are included in the cutset. That is, suppose that the event is represented by the string

$$W = \texttt{021111221111111111121121}, \tag{2-6a}$$

which indicates that the link numbered N+1 is specified in the event to be DOWN and the links N+2, N+7, N+8, N+21, and N+24 are specified to be UP. Further, suppose that the encoded cutset is

$$\textit{UpEdges} = \texttt{111112111111111121111111}, \tag{2-6b}$$

which indicates that the cutset consists of the failures of links N+6 and N+18. Then the failure event is determined by the procedure *CutFail* in CUTONLY to be

$$F = \texttt{021110221111111101121121}. \tag{2-6c}$$

Unlike the link sequence order for paths, the order of the listing of the links whose failures are in the cutset has not been found to be significant in affecting the number of next events, so they are entered into *PathList* in their lexicographical order. Another aspect of the processing of cutsets that is different from that of pathfinding is that the number of new events is always equal to the number of links listed in *PathList*, since by definition any link specified in the event to be DOWN will not be in the cutset.

The s-t reliability calculation in EL2ONLY is truncated in the same manner as that in EQLNKTST.

2.2.2.3 Program combining pathfinding and cutsets

In order to exploit the fact that the upper bound converges faster than the lower bound for the ELA2, while the opposite is true for the ELA, the program EL1&2 that is listed in Appendix D.3 was developed, using procedures and functions in the unit NETSET and in the unit CUTSET (which also is listed in Appendix D.3).

Essentially, EL1&2 implements *both* the ELA and the ELA2, using two sets of queues, with the truncation based on the convergence of the lower bound from the pathfinding method and the upper bound from the cutset-finding method. The two parts of the program operate as described in Sections 2.2.2.1 and 2.2.2.2.

2.2.2.4 Program selecting pathfinding or cutsets

It was noted in Section 1.2.3 that the partitioning of events can alternate between the pathfinding method and the cutset-finding method. The program ELCUTPAT that is listed in Appendix D.4 (along with its supporting unit, CUTPATH) was written to combine features of the ELA and the ELA2. Like the other programs for calculating s-t reliability, it utilizes procedures and functions in the unit NETSET.

In ELCUTPAT, a single pair of queues is used, and the overall design philosophy of the program is to minimize the number of new events that are generated, by choosing either the pathfinding or the cutset-finding approach. As illustrated by the partial flow diagram in Figure 2-8, after reading an event to be tested the program first seeks an $s \rightarrow t$ path. If one is not found, the event is definitely a failure, which does not generate a new event; the event is therefore processed as an ELA failure event and the next event to be tested is read from the input queue.

If a path is found, a determination is made of *NewCount*, the number of new events that would be generated if the event is processed as an ELA success event. If *NewCount* is 0, then no new events will be generated because all the links in the path were specified as being UP in the event being tested; this value of *NewCount* is possible if the event being tested was previously generated using the cutset approach. If *NewCount* is 1, then a cutset exists but it cannot produce any fewer new events. Therefore, if *NewCount* < 2, preference is given to pathfinding: the event is processed as an ELA success event.

If *NewCount* ≥ 2, then there is a cutset, possibly with the number *CutCount* of link failures in it such that *CutCount* $<$ *NewCount*, giving rise to fewer new events if the cutset approach is used. To test for this condition, a cutset is found. If *CutCount* $<$ *NewCount*, the event being tested is processed as an ELA success event; otherwise, it is processed as an ELA2 failure event.

It is demonstrated in Appendix C for an example network that this methodology yields a total of 30 events into which the probability space is partititioned (11 successes and 19 failures). When the same example network was analyzed using the ELA, a total of 38 events were found (11 successes and 27 failures); using the ELA2, a total of 33 events were found (14 successes and 19 failures). For this example, the “cutpath" method did as well as the ELA in obtaining relatively large success partitions and as well as the ELA2 in obtaining relatively large failure partitions.

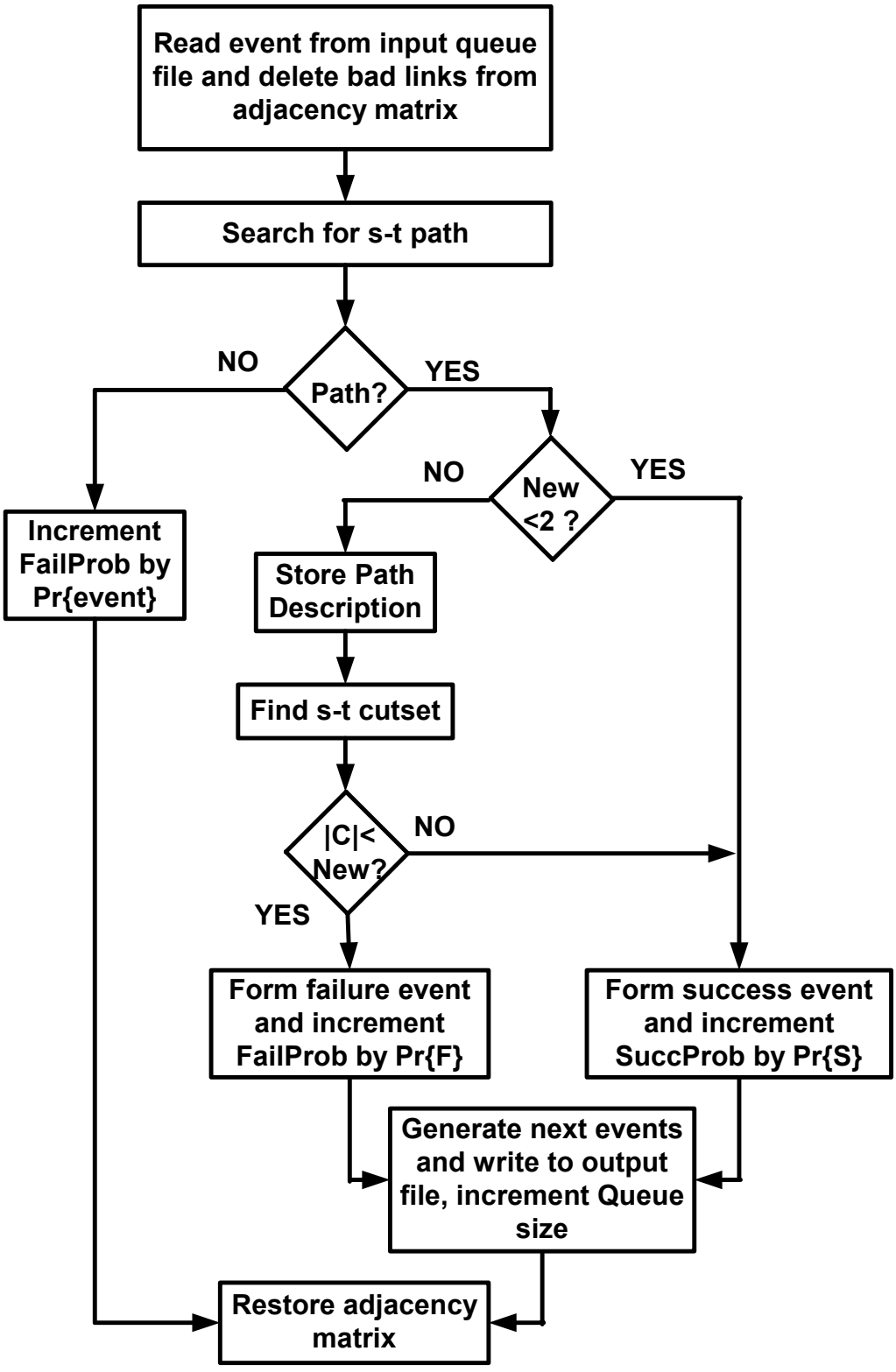


FIGURE 2-8 FLOW FOR SELECTING PATHS OR CUTSETS

Figures 2-9 and 2-10 show, respectively, how the upper and lower bounds based on partial sums of success and failure probabilities for the cutpath method compare with those for the ELA and the ELA2, and how the accuracy of an estimate of the s-t reliability based on averaging the cutpath bounds compares with that of the other estimates considered. Both figures indicate that the cutpath method has the potential of improving the convergence and accuracy of the bounds and estimates based on them.

Truncation in ELCUTPAT is based on the same criteria as in the other programs. The execution of ELCUTPAT is faster than that of EL1&2 because (1) queue-reading I/O operations are fewer and (2) pathfinding and cutset-finding are not always done for a given event. On the other hand, a direct comparison of the two programs is difficult because the evolution of the partitioning of the probability space is different in them.

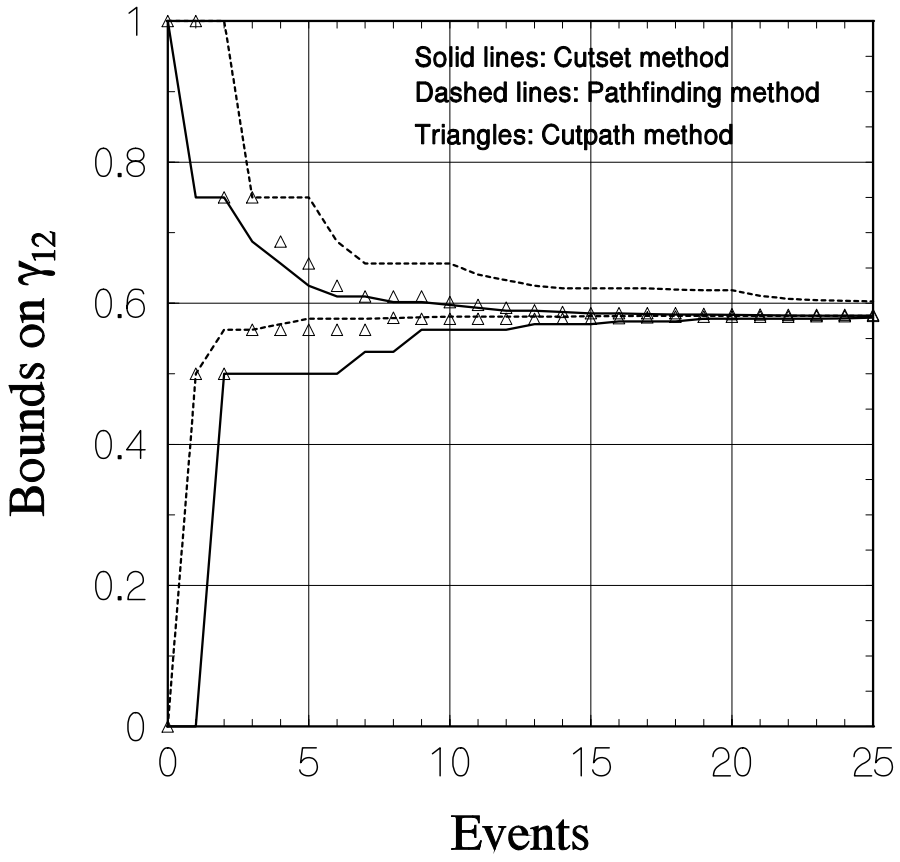


FIGURE 2-9 COMPARISON OF BOUNDS WITH CUTPATH BOUNDS

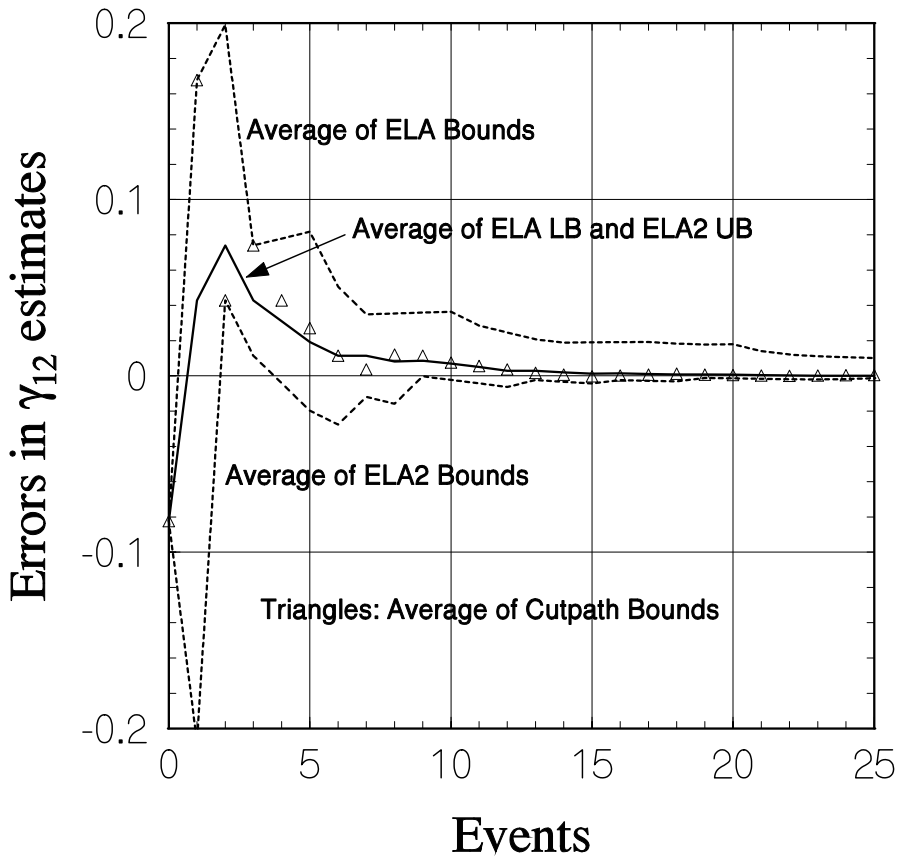


FIGURE 2-10 COMPARISON OF ERRORS WITH CUTPATH ERROR

2.2.3 Programs Based on Factoring

An alternative method for calculating network s-t reliability, given a source node s and a destination (or sink) node t, is to use an algorithm described by Page and Perry [9]. The algorithm is based on an application of the network factoring theorem to networks with directed edges that may fail and vertices (nodes) that do not fail. The network factoring theorem may be expressed by

$$\gamma_{st}(G) = \beta_l \cdot \gamma_{st}(G{*}l) + (1 - \beta_l) \cdot \gamma_{st}(G{-}l), \tag{2-7a}$$

where G denotes a given network or graph, described completely by a list of links (or directed edges) and their reliabilities, where l denotes a selected link, and

$$G{*}l \triangleq \text{network } G \text{ with link } l \text{ contracted} \tag{2-7b}$$

$$G{-}l \triangleq \text{network } G \text{ with link } l \text{ deleted.} \tag{2-7c}$$

By “contraction" is meant the elimination of a link by merging the link's source and destination nodes; for directed networks, such contraction requires that there not be another link antiparallel to link l, joining the same two nodes but going in the opposite direction. This requirement is easily met by selecting a link l from among the links going out of node s or into node t, since any links antiparallel to these links can be “pruned" (deleted) from the network without changing its s-t reliability.

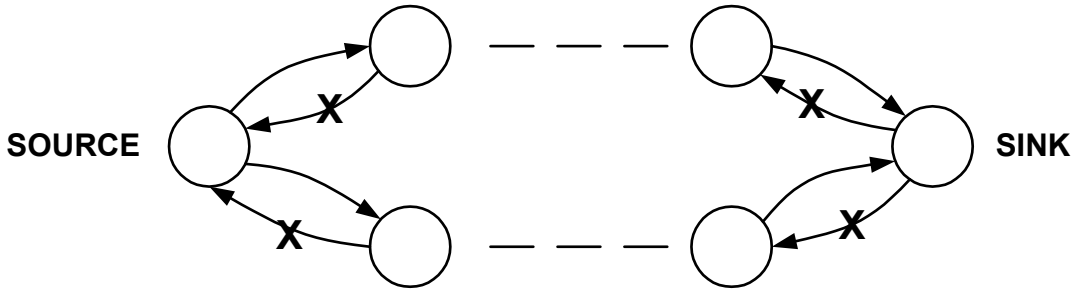
The Page-Perry algorithm is recursive. Its operation therefore may be represented by the following expansion of (2-7a):

$$\begin{aligned}\gamma_{st}(G) &= \beta_1 \cdot \gamma_{st}(G{*}l_1) + (1-\beta_1)\cdot\gamma_{st}(G{-}l_1)\\ &= \beta_1\cdot\{\beta_2\cdot\gamma_{st}[(G{*}l_1){*}l_2] + (1-\beta_2)\cdot\gamma_{st}[(G{*}l_1){-}l_2]\}\\ &\qquad + (1-\beta_1)\cdot\{\beta_2\cdot\gamma_{st}[(G{-}l_1){*}l_2] + (1-\beta_2)\cdot\gamma_{st}[(G{-}l_1){-}l_2]\}\\ &= (\beta_1\beta_2\beta_3\cdots)\gamma_{st}[G{*}l_1{*}l_2{*}l_3{*}\cdots] + \cdots. \end{aligned} \tag{2-8}$$

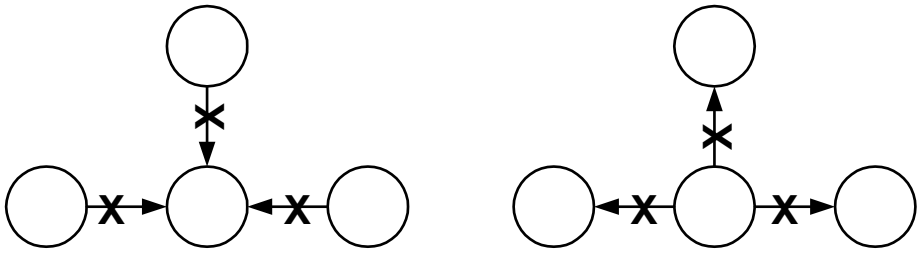
It is evident that the recursion, unmodified, would in effect generate the probabilities of the 2^M network elementary events, where M is the number of links, and so is an alternative to other ways of accounting for these events, such as the equivalent-links algorithm, but not very efficient. However, in [9] a great efficiency is reported for this recursive algorithm, derived from simplifying the network at each step by pruning and reduction techniques, therefore reducing the dimensionality of the problem.

Pruning techniques that can be used to simplify a network and to reduce the dimensionality of its analysis problem are illustrated in Figure 2-11. Basically they recognize the links that are “irrelevant" to the s-t reliability because they can be predicted not to carry any message flow from source to sink. Such links can be deleted from the list of links describing the network. Also, nodes that do not relay messages from the source to the sink are irrelevant and may be removed, along with the links into or out of them.

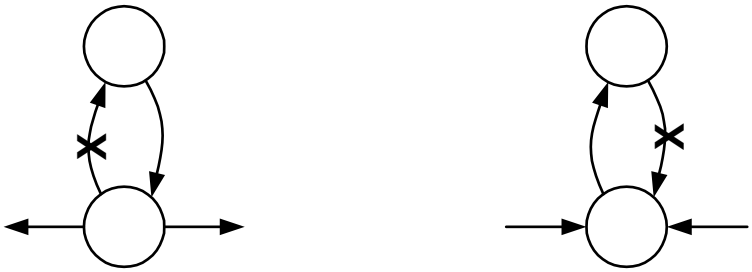
In addition to pruning, reduction techniques can be used to simplify the network by recognizing how certain configurations of nodes and links can be replaced by simpler configurations with the same effective reliability. Various reduction techniques are illustrated in Figure 2-12. All of these are implemented in the Page-Perry algorithm except the one labelled (b) in Figure 2-12, which is said not to be needed because of the eventual application of one or more of the pruning techniques as the algorithm proceeds.



(a) Links into source or out of sink are irrelevant



(b) Nodes with only inputs or only outputs (except source and sink) are irrelevant



(c) Links antiparallel to a node's only input or output are irrelevant

FIGURE 2-11 PRUNING TECHNIQUES THAT CAN BE USED TO SIMPLIFY A NETWORK ANALYSIS PROBLEM

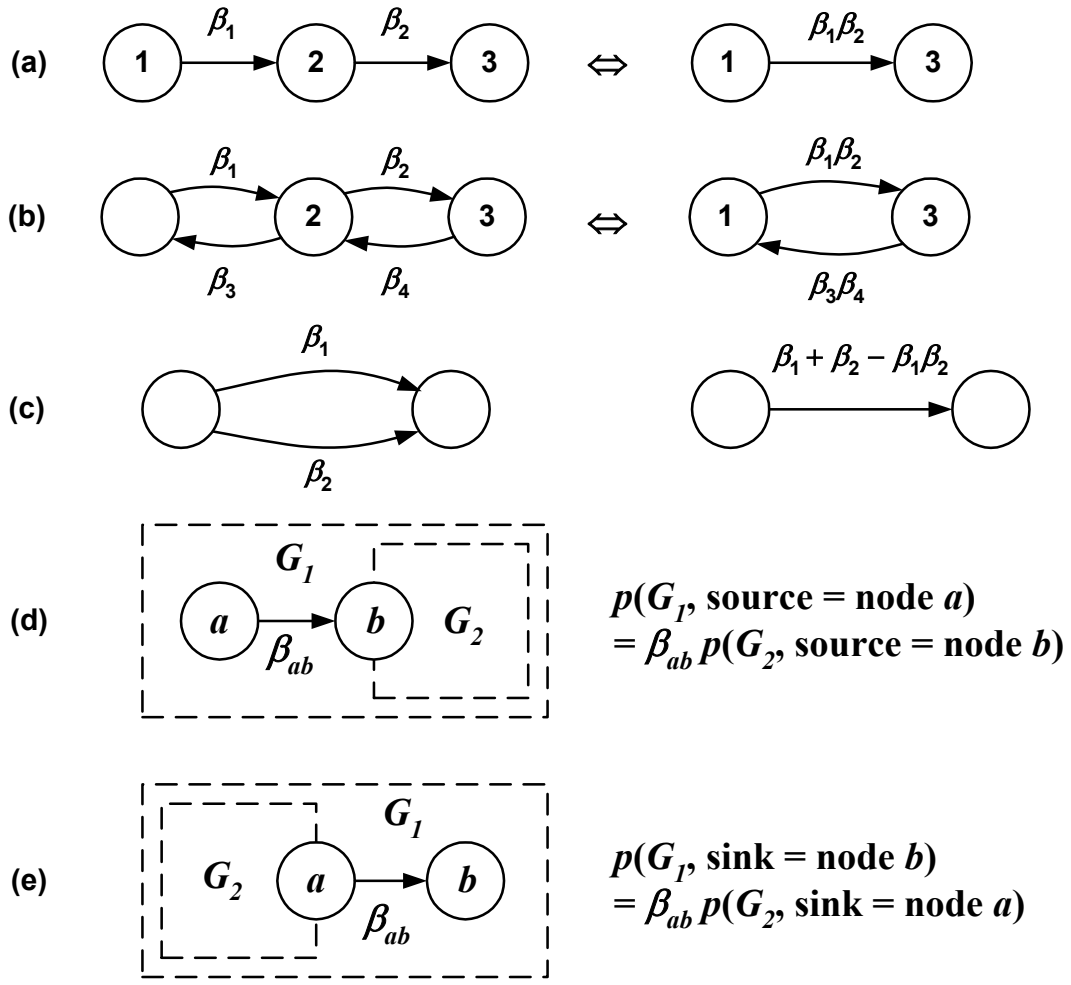


FIGURE 2-12 NETWORK REDUCTION TECHNIQUES USED BY THE PAGE-PERRY ALGORITHM

The Theologou-Carlier algorithm [10] for computing *s-t* reliability is similar to the Page-Perry algorithm but allows for nodes that may fail. The authors note, in effect, that the pruning techniques shown in Figure 2-11 are valid for networks with imperfect nodes, and that the simple reduction techniques shown in Figure 2-12 are easily modified to account for imperfect nodes. The more serious modifications to the Page-Perry algorithm have to do with accounting for the necessary changes in node reliabilities when the factoring theorem is applied.

The authors of [10] note that the factoring theorem can be generalized to

$$\gamma_{st}(G) = p_l \cdot \gamma_{st}(G{*}l) + (1 - p_l) \cdot \gamma_{st}(G{-}l), \tag{2-9}$$

where l now represents successful transmission between two failable nodes over a failable link. Let the nodes and the link be labelled u, v, and e, respectively. Since l working implies that all three of u, v, and e are working (with probability $p_l = \alpha_u\beta_e\alpha_v$), then "$G{*}l$" denotes the graph with e contracted and the combined node u-v having reliability 1. Since l not working implies one or more of u, v, and e not working (with probability $1 - p_l = 1 - \alpha_u\beta_e\alpha_v$), then "$G - l$" denotes the graph with e deleted and with u and v remaining, but with modified reliabilities

$$\alpha'_u = \tfrac{\alpha_u(1-\beta_e\alpha_v)}{1-\alpha_u\beta_e\alpha_v} \text{ and } \alpha'_v = \tfrac{\alpha_v(1-\beta_e\alpha_u)}{1-\alpha_u\beta_e\alpha_v}\,. \tag{2-10}$$

Note that, in the graph $G - l$, failures of the nodes u and v are no longer independent. This difficulty is circumvented by taking note of the facts that (1) the s-t reliability of G is $\alpha_s\alpha_t$ times the s-t reliability of G when it is assumed that $\alpha_s = \alpha_t = 1$; and (2) if node u, the source of e, does not fail ($\alpha_u = 1$) then the modified reliability of node v in (2-10) no longer depends on the reliability of node u:

$$\alpha'_v = \tfrac{\alpha_v(1-\beta_e)}{1-\beta_e\alpha_v}\,. \tag{2-11}$$

These facts taken together suggest an iterative procedure for finding the s-t reliability (with α_s factored out) in which the first edge to be factored on is one of the edges out of the source node, or (with α_t factored out) in which the first edge to be factored on is one of the edges out of the sink node. The pruning and reductions as the iterations of (2-9) proceed are entirely equivalent to those of Page and Perry, with the slight additional complexity of modifying the node reliabilities appropriately in the manner we have shown above.

2.2.3.1 Program implementing the Theologou-Carlier algorithm

The program TCPTR listed in Appendix D.6 implements the Theologou-Carlier algorithm (TCA) for calculating s-t reliability. As listed, the program contains procedures and functions to facilitate the user interface, and makes use of other procedures and functions residing in the TurboPascal unit TCUPTR, which also is listed in Appendix D.6.

Prior to the reliability calculation, TCPTR calls the procedure *BuildGraph*, which creates a network description from the information supplied by the user through keyboard entry and SNR values for the network links in the *.SNR file specified by the user. The network description is in the form of a special data structure including the following components:

```
Graph = RECORD                    {Describes a graph}
    Vert : GraphSet;              {Set of graph vertices}
    Source,                       {Source vertex}
    Sink : Integer;               {Sink vertex}
    InDegree,                     {In degree of each vertex}
    OutDegree : DegreeType;       {Out degree of each vertex}
    nb : ARRAY[1..NODEMAX] OF GraphSet;
                                  {Edge(i,j) puts j in nb[i]}
    NumEdges : Integer;           {No. of edges in the graph}
    NumNodes : Integer;           {Largest-numbered vertex}
    e : ARRAY[1..EDGEMAX] OF Edge;
                                  {Describes edges in graph}
    Alpha : ARRAY[1..NODEMAX] OF Real;
                                  {Node reliabilities}
  END; { Graph }                                        (2-12a)
```

where

```
DegreeType = ARRAY[1..NODEMAX] OF Integer;
                                  {List of vertex degrees}
GraphSet = SET OF 1..NODEMAX;     {Set of vertices}
Edge = RECORD                     {Edge in a graph}
    Start,                        {Start vertex}
    Stop : 1..NODEMAX;            {Stop vertex }
    Beta : Real                   {Edge reliabilities}
  END; { Edge }                                         (2-12b)
```

In these Pascal statements, it is assumed that maximum numbers of vertices (nodes) and edges (links) have been defined as constants; the program necessarily oversizes the arrays because they cannot be changed dynamically in Pascal.

In order to understand how the program operates, it is only necessary to realize that for a particular case, in the mathematical notation used in Section 1, there are N nodes and M links. So the network data structure described in the statements above consists essentially of a set (= *Vert*) containing the numbers (labels) of the links in the network; s (= *Source*); t (= *Sink*); a list of the number of links entering each node (= *InDegree*); a list of the number of links leaving each node (= *OutDegree*); a collection of N sets (= *nb*) containing the numbers (labels) of the other nodes to which each node is connected by a link leaving that node; M (= *NumEdges*); N (= *NumNodes*); a list of

node reliabilities ($=$ *Alpha*); and a list ($= e$) of the start node, stop node, and reliability for each link.

The procedure *BuildGraph* examines the SNRs for all $N(N-1)$ possible links and skips links that practically speaking don't exist, that is, those for which the link margin is less than zero, giving rise to a β_{ij} that is less than $\mathrm{P}_G(0) = 0.5$. Those links with $\beta_{ij} \geq 0.5$ are considered viable and are entered into the data structure: each is numbered (labelled); its start node, stop node, and reliability are entered into an *edge* record; the nodes associated with the edge are added to the set *Vert* and the maximum node number *NumNodes* is updated, if necessary; and the number (label) of the edge's ending node is added to the set of nodes reached by the edge's starting node.

Having embedded the information on the network in this directed graph data structure referenced by the variable name g, the program finds the input and output degrees of each node and computes the s-t reliability as

$$\gamma_{st} = \mathit{Prob}(g). \tag{2-12}$$

The function *Prob*(g) has the flow diagrammed in Figure 2-13. Upon initiation of the function, with the variable g passed to it as the argument, various pruning and reduction procedures are applied to g in order to simplify the graph. Then the source and sink node reliabilities are factored out, leaving a graph whose source and sink nodes do not fail. This factoring permits application of the Theologou-Carlier version of the network factoring theorem stated above in (2-9) and (2-11), using an edge into the sink. The program finds such an edge (l) and the node from which it originates (q), and implements the expression

$$\mathit{Prob}(g) = \alpha_s \alpha_t \left[\beta_l \alpha_q \mathit{Prob}\left(g' * l\,;\, \alpha'_q = 1\right) + (1 - \beta_l \alpha_q) \mathit{Prob}\left(g' - l\,;\, \alpha'_q = \tfrac{\alpha_q(1-\beta_l)}{1-\beta_l \alpha_q}\right) \right] \tag{2-13}$$

in which g' denotes the graph g after pruning and reduction. Note in (2-13) that the function is recursive. Each time *Prob* is called from within itself, copies of the variables involved in the calculation using the call have to be stored; if the depth of recursion is too great, the computer can run out of “stack" memory. For that reason, the Pascal code implementing *Prob* uses pointer variables to reference the graph data structure, allowing the memory to be used as needed (not from the stack, but from the larger portion of the computer's memory allocated to the “heap") and released when not needed.

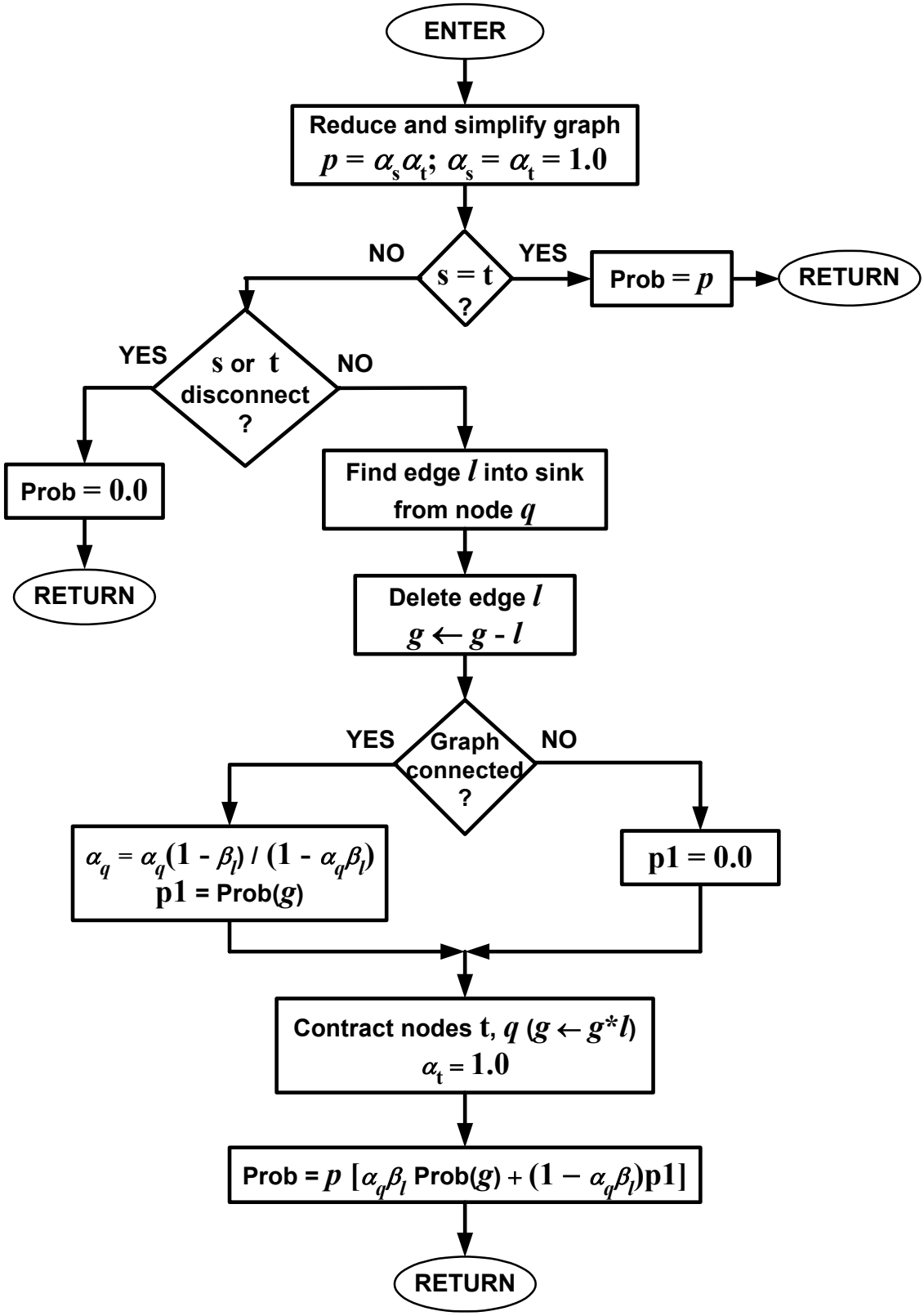


FIGURE 2-13 FLOW IMPLEMENTING THE FUNCTION *Prob*(*g*)

2.2.3.2 Program implementing the TCA with bounds

For relatively large networks, the program TCPTR implementing the TCA can take an excessively long time to execute. For that reason, modifications to the algorithm were developed that permit the fast computation of upper and lower bounds, and these bounds have been implemented in the program TCPTRBND that is listed in Appendix D.7, supported by the unit TCUPTRUL.

In the notation of (2-9), the upper bound is based on the observation that

$$\gamma_{st}(G) = p_l \cdot \gamma_{st}(G^*l) + (1 - p_l) \cdot \gamma_{st}(G-l)$$

$$\leq \gamma_{st}(G \text{ with } p_l = 1.0) = 1.0 \times \gamma_{st}(G^*l) + (1 - 1.0) \times \gamma_{st}(G-l), \quad (2\text{-}14a)$$

or $$\gamma_{st}(G) \leq \gamma_{st}(G^*l). \quad (2\text{-}14b)$$

This bound results in a calculation with potentially much fewer recursions, and is used when the program finds that β_l , the reliability of the link l [in the notation of (2-13)], is greater than a probability threshold[5]; the higher the value of that threshold, the tighter the bound. A lower bound[6] may be computed similarly from the relationship

$$\gamma_{st}(G) \geq p_l \cdot \gamma_{st}(G^*l) + 0 \cdot \gamma_{st}(G-l)$$

or $$\gamma_{st}(G) \geq p_l \cdot \gamma_{st}(G^*l), \quad (2\text{-}15)$$

ignoring terms of the form $(1 - p_l) \times \Pr\{\text{subgraph}\}$ when a particular link reliability is close to unity. The tightness of this bound, invoked when β_l exceeds a probability threshold, depends upon the value of that threshold.

2.2.4 A Program with Combined Partitioning and Factoring

In a recent paper in the *IEEE Transactions on Reliability* [11], an algorithm was introduced that makes uses of both factoring (including reductions and simplifications of the network) and the partitioning concepts employed by the ELA. Given the nodes s and t, plus an $s \rightarrow t$ path consisting of the sequence of links

$$P = (l_1 , l_2 , l_3, \ldots , l_K), \quad (2\text{-}16)$$

the factoring theorem expressed generally above in (2-8) can be put into the following form by grouping terms:

---

[5]A discussion of the heuristic probability thresholds that are used is given later in this report, in connection with the numerical results.

[6]Another, simple lower bound is $\alpha_s\alpha_t\beta_{st}$, the reliability of the direct path $s \rightarrow t$.

$$
\begin{aligned}
\gamma_{st}(G) &= (1-\beta_1)\cdot\gamma_{st}(G{-}l_1) + \beta_1\cdot\gamma_{st}(G{*}l_1) \\
&= (1-\beta_1)\cdot\gamma_{st}(G{-}l_1) + \beta_1\{(1-\beta_2)\cdot\gamma_{st}[(G{*}l_1){-}l_2] + \beta_2\cdot\gamma_{st}(G{*}l_1{*}l_2)\} \\
&= (1-\beta_1)\cdot\gamma_{st}(G{-}l_1) + \beta_1\cdot(1-\beta_2)\cdot\gamma_{st}[(G{*}l_1){-}l_2] \\
&\qquad + \beta_1\beta_2\{(1-\beta_3)\cdot\gamma_{st}[(G{*}l_1{*}l_2)-l_3] + \beta_3\cdot\gamma_{st}(G{*}l_1{*}l_2{*}l_3)\} \\
&= (1-\beta_1)\cdot\gamma_{st}(G{-}l_1) + \beta_1\cdot(1-\beta_2)\cdot\gamma_{st}[(G{*}l_1){-}l_2] \\
&\qquad + \beta_1\beta_2(1-\beta_3)\cdot\gamma_{st}[(G{*}l_1{*}l_2)-l_3] \\
&\qquad \vdots \\
&\qquad + \beta_1\beta_2\beta_3\cdots\beta_{K-1}(1-\beta_K)\cdot\gamma_{st}[(G{*}l_1{*}l_2{*}l_3{*}\cdots{*}l_{K-1})-l_K] \\
&\qquad + \beta_1\beta_2\beta_3\cdots\beta_{K-1}\beta_K\cdot\underbrace{\gamma_{st}(G{*}l_1{*}l_2{*}l_3{*}\cdots{*}l_{K-1}{*}l_K)}_{=1}\,.
\end{aligned}
\tag{2-17}
$$

The terms in the last equation above correspond to partitioning the 2^M elementary events into the $K+1$ disjoint events

$$
\begin{array}{lcll}
 & & \overbrace{}^{\text{links }1\ldots K}\overbrace{}^{K+1\ldots M} & \\
\bar{l}_1 & \text{or} & 0\,x\,x\,x\cdots x\,x\;x\,x\cdots x & (2^{M-1}\text{ elem. events}) \\
l_1\bar{l}_2 & \text{or} & 1\,0\,x\,x\cdots x\,x\;x\,x\cdots x & (2^{M-2}\text{ elem. events}) \\
l_1l_2\bar{l}_3 & \text{or} & 1\,1\,0\,x\cdots x\,x\;x\,x\cdots x & (2^{M-3}\text{ elem. events}) \\
\vdots & & \vdots & \\
l_1l_2l_3\cdots l_{K-2}\bar{l}_{K-1} & \text{or} & 1\,1\,1\,1\cdots 0\,x\;x\,x\cdots x & (2^{M-K+1}\text{ elem. events}) \\
l_1l_2l_3\cdots l_{K-2}l_{K-1}\bar{l}_K & \text{or} & 1\,1\,1\,1\cdots 1\,0\;x\,x\cdots x & (2^{M-K}\text{ elem. events}) \\
l_1l_2l_3\cdots l_{K-2}l_{K-1}l_K & \text{or} & 1\,1\,1\,1\cdots 1\,1\;x\,x\cdots x & (2^{M-K}\text{ elem. events})\,.
\end{array}
\tag{2-18}
$$

The probability of the last subgraph in (2-17) is 1 because all the links in the path P are postulated to be UP.

As written, (2-17) is simply one of many ways to group the 2^M terms of (2-8) into $K+1$ terms, since it is understood that each use of the recursive function $\gamma_{st}(\,\cdot\,)$ involves an expansion and further recursions (after network reductions and simplifications). However, there is an advantage to the grouping shown in (2-18): as noted in [11], each subgraph has at least one less $s \rightarrow t$ path than the original graph because one of the links in P is postulated to have failed. Another insight comes from noting that for each successive subgraph, there is at least one less link and one less node, since contracting a link merges two nodes. In [11] it was reported that this “reduce and partition" algorithm typically computes γ_{st} in about half the time that is required by the original factoring algorithm with reduction [9] that was discussed in Section 2.2.3.

The program REDNPART that is listed in Appendix D.8.1 implements the reduce and partition algorithm as a modification to the Theologou-Carlier algorithm for computing the exact s-t reliability with nodes that can fail. The program RNPBOUND that is listed in Appendix D.8.2 utilizes the same kind of bounding approach that was discussed in Section 2.2.3.2 in order to obtain upper and lower bounds for the reliability.

2.3 ALGORITHM PERFORMANCES

In this subsection, the performances of the various s-t reliability algorithms described above are presented. Algorithm performance is given both in terms of accuracy and of time required to compute either exact reliabilities or bounds for selected example networks.

Side-by-side tests of the algorithms were performed in several stages. The initial tests were used to establish a general ranking of algorithm performance for a variety of network sizes. Certain tests, summarized in Appendix E, were conducted to develop refinements to the more promising algorithms. In addition to showing the degree of improvement that can be achieved by efficient programming, these tests established that the best performance of the partitioning class of algorithms in terms of execution time generally is experienced for two-way pathfinding with anti-return logic; the time is not significantly affected by the choice between the cutset search methods using the first cutset found or using the larger cutset found. The studies summarized in Appendix E also revealed that the EL2ONLY program, which uses only cutsets to develop reliability events, requires an excessive amount of program execution time for large networks. Therefore, this program was eliminated from the comparisons presented in this subsection.

The algorithms were tested using the following *.SNR files:

- MESH1000.SNR. (Used for the tests reported in Appendix E only.) This file contains SNRs for the links of the 3×3 example network shown in Figure 1-1 and discussed in Section 2.1.1, with the particular (s, t) pair being (1, 2). The link reliabilities for this case are listed in Table 2-1. In addition to being an example for which there is an analytical solution (see Appendices A, B, and C), this case runs very quickly for any algorithm.

- T920811.SNR. This file contains SNRs for the links of the 15-node example network shown in Figure 2-1 and discussed in Section 2.1.2, with the particular (s, t) pair being (8, 13). The link reliabilities for this case are listed in Table 2-2. This case repre-

sents a moderately challenging network example in which there is no one-hop $s \rightarrow t$ path.

- BIGONE.SNR. This file contains SNRs for the links of the 34-node example network shown in Figure 2-2 and discussed in Section 2.1.3, with a focus on the particular (s, t) pair (25, 20), which has a minimum hop distance of 6. This example represents a realistic-sized MSE network, and was studied in [1].

2.3.1 Comparisons of Exact Calculations

The two programs implementing exact calculations of s-t reliability are TCPTR, which uses the Theologou-Carlier (factoring) approach discussed in Section 2.2.3.1, and REDNPART, which uses the combined reduction and partition approach discussed in Section 2.2.4. The results of the calculations are as follows:

Case	Result	Time, TCPTR	Time, REDNPART
3 × 3, (1, 2)	.93133093	$\ll$ 1 s	$\ll$ 1 s
15-node, (8, 13)	.92982718	49 s	24 s
34-node, (25, 20)	----------	$\gg$ 14 hrs	$\gg$ 14 hrs

The exact answer for the 34-node example network was not available using either algorithm after having run for 14 hours.

The small time for the 3 × 3 example network is not surprising. It is perhaps surprising that the run time for the 15-node example network, though not excessive, is so much greater than that for the 3 × 3 network. The difference is only partially accounted for by the size of the networks: the 15-node network has 60 links, whereas the 3 × 3 has 24 links; $(49)^{24/60} = 4.7$ s would be the case for the 3 × 3 network, assuming an exponential dependence of the time of the TCPTR program upon the number of links. The "reducibility" of the examples undoubtedly has something to do with the relative times, but an independent measure of such a characteristic is not available—rather, the times themselves would seem to be indicators that could be used to develop a measure of reducibility.

2.3.2 Comparisons of Bound Calculations

2.3.2.1 The programs that are compared

Five different programs were used to calculate upper and lower bounds on the s-t reliability for the example cases.

The three equivalent-links programs (EQLNKTST, EL1&2, and ELCUTPAT) are based on partitioning, and utilize a stopping criterion that “tracks" the convergence of the bounds: the program is terminated when UB − LB < 0.01 and at least 98 events have been processed, unless the calculation is exact for fewer events processed. In addition, EL1&2 stops if either the ELA or the ELA2 portion finds the exact s-t reliability before the bounds converge. The baseline configuration of these algorithms for the results presented includes the following choices of search techniques:

- Pathfinding: two-way flood search with prevention of return to any search-propagating node (anti-return logic). When the search at a given stage (hop) reaches a particular node from two or more previously reached nodes, an arbitrary preference is given to the previously reached node having the lowest number (label). For all the results in this report, a 10-hop path limit was imposed. In Appendix E, a study is made of the effect of other pathfinding approaches on the performances of the programs.

- Cutset Search: two-way cutset search with selection of the first cutset that is found. In Appendix E, it is shown that the selection of the larger cutset, as discussed in Section 1.2.2, is neither uniformly better nor worse in terms of execution time.

Of the other two programs, TCPTRBND is based on reduction and factoring while RNPBOUND is based on a combination of reduction and partitioning; both of these algorithms do not track bound convergence, and therefore could be termed “open-loop." These programs implement new concepts for obtaining upper and lower bounds on the s-t reliability that were not included in the original algorithms. For the results presented in this section, adaptive probability thresholds were used to implement the bounding techniques explained in Section 2.2.3.2, as discussed below. Also, the lower and upper bound calculations were combined using one exercise of the basic Theologou-Carlier algorithm for each subgraph to avoid duplicate calculations.

The adaptive threshold used may be expressed by

$$p_{th} = \beta_{max} - k \cdot \delta \,, \tag{2-19a}$$

where

$$\delta = (\beta_{max} - \overline{\beta})/\min\{100(1.01 - \overline{\beta}),\, 26\} \tag{2-19b}$$

is an adaptive step size and where k is an integer indexing the depth of recursion. The smaller the value of p_{th} , the more likely it is that the program will use an approximation for the probability of the subgraph, thereby running faster. Note that as k increases, the probability threshold decreases gradually from β_{max} to $\overline{\beta}$. For $\overline{\beta} > 0.75$, p_{th} decreases to

$\overline{\beta}$ relatively quickly, in 10 to 26 recursions. Otherwise p_{th} decreases to $\overline{\beta}$ relatively slowly, in 26 recursions.

When the program first calls the probability calculation procedure, the depth of recursion is $k = 0$; each time that procedure calls itself to carry out the network factoring, k is incremented, causing the probability threshold to decrease. This thresholding scheme, used for both the lower and upper bound calculations, forces at least one recursion since for $k = 0$ none of the link reliabilities exceed the threshold. For $k > 0$, factoring on a link with high reliability can result in an approximation—leading either to an upper bound or to a lower bound, as discussed in Section 2.2.3.2, by neglecting the probability of the subgraph with the factored link deleted. As the depth of the recursion increases, it is more likely that an approximation is used. By this heuristic method, the depth of recursion is made self-limiting and the overall execution time of the algorithm is contained.

Developmental tests of the TCPTRBND program revealed that the lower bound (2-15) developed by the Theologou-Carlier type of algorithm tends to be loose; this tendency is especially pronounced for SNR conditions giving rise to sets of relatively low link reliability values. The reason for the looseness is that the neglected term has been neglected on the basis that it is multiplied by $(1 - p_l)$, which is small for $p_l \approx 1$. However, for low SNR conditions, p_l is not close to 1. Therefore, in addition to implementing the adaptive threshold given in (2-19), the versions of RNPBOUND and TCPTRBND[7] used for the parametric performance results that follow employs an additional heuristic: the lower bound is computed as

$$\underbrace{\gamma_{st}(G) \geq p_l \cdot \gamma_{st}(G^{*}l)}_{\text{LB as expressed previously}} + \underbrace{(1 - p_l) \cdot \Pr\{\text{shortest } s \rightarrow t \text{ path for } G - l\}}_{\text{additional term to tighten LB}}. \qquad \text{(2-20)}$$

Tests of algorithm performance were made using the files T920811.SNR and BIGONE.SNR for the 15-node and 34-node example networks, respectively, in order to show the parametric variation in the performances of the algorithms as a function of the SNR conditions. As discussed in Section 2.2.1, the first task performed by each of the programs implementing the s-t reliability algorithms is to calculate link reliabilities using the selected link SNR data. In order to generate sets of performance results parametric in

[7]In Appendix E, fixed link probability thresholds of 0.9 and 0.96, respectively, are used for calculating the TCPTRBND upper and lower bounds as given by (2-14) and (2-15). A higher threshold value was used to calculate the lower bound because this bound was found to be looser than the upper bound for the same value of threshold. Also, lower and upper bounds were calculated by two separate exercises of the basic Theologou-Carlier algorithm.

the stress conditions affecting the networks, it therefore was expedient to vary the value of SNR threshold entered from the keyboard, simulating the variation of the SNR for a fixed threshold. Mathematically, this simulation can be expressed by

$$\beta_{ij}(n) = \mathrm{P}_G\left[\frac{\mathrm{SNR}_{ij} - (n + \rho_0)}{\sigma_L}\right] = \mathrm{P}_G\left[\frac{(\mathrm{SNR}_{ij} - n) - \rho_0}{\sigma_L}\right], \tag{2-21}$$

in which $\mathrm{P}_G(\,\cdot\,)$ denotes the Gaussian cumulative probability distribution function, σ_L is the standard deviation of the SNR (taken to be $\sigma_L = 10$ dB), and incrementing the SNR threshold ρ_0 by n dB is seen to be equivalent to decrementing the SNR by n dB. The value $\rho_0 = 0$ dB is used throughout this study, giving a link reliability of 0.50 when there is a link margin of zero dB.

2.3.2.2 Tests for the 15-node example network

Parametric calculations were made for the 15-node example network with $(s, t) =$ (8, 13). Tests for the 34-node example network are discussed in Section 2.3.2.3.

Figure 2-14 shows the upper and lower bounds calculated by five programs: EQLNKTST, EL1&2, ELCUTPAT, TCPTRBND, and RNPBOUND. All of the upper bounds are plotted with a solid line, and all of the lower bounds, with a dashed line. In this case, since the 10-hop limit on paths did not significantly constrain the probability that was calculated using the equivalent-links programs, there is close agreement with the bounds calculated by the recursive programs. Also, by chance (since there is no mechanism to control their accuracy) the recursive programs' bounds are at least as tight as those calculated by the equivalent-links programs (see Figure 2-17, below).

The steep drop in the value of the probability in going from a threshold of 4 dB to one of 5 dB is due to the fact that one of the two links emanating from the source node (node 8) goes from an UP to a DOWN status and is eliminated as its reliability becomes less than 0.5.

In terms of execution time, Figure 2-15 shows the relative performances of the five programs; note the logarithmic scale, necessary because of the wide range in times. As the threshold is increased, changing the connectivity, the speed ranking of the four programs changes, but overall the programs based on factoring and reduction tend to be the fastest for this example. Note that the EL1&2 program is much slower than the ELA for this example, while ELCUTPAT is at most slower by a factor of two.

For the three equivalent-links programs, generally the execution time increases with the threshold when it is less than 4 dB, and decreases when the threshold is greater than 4 dB; this behavior is caused by the increased number of success events needed to accumulate the lower bound when the link reliabilities are not high—for a constant connectivity. As noted in connection with Figures 2-14, for 5 dB and higher the supply of paths for the (s, t) pair is small, giving rise to a more rapid accumulation of the lower bound, though it has a smaller value.

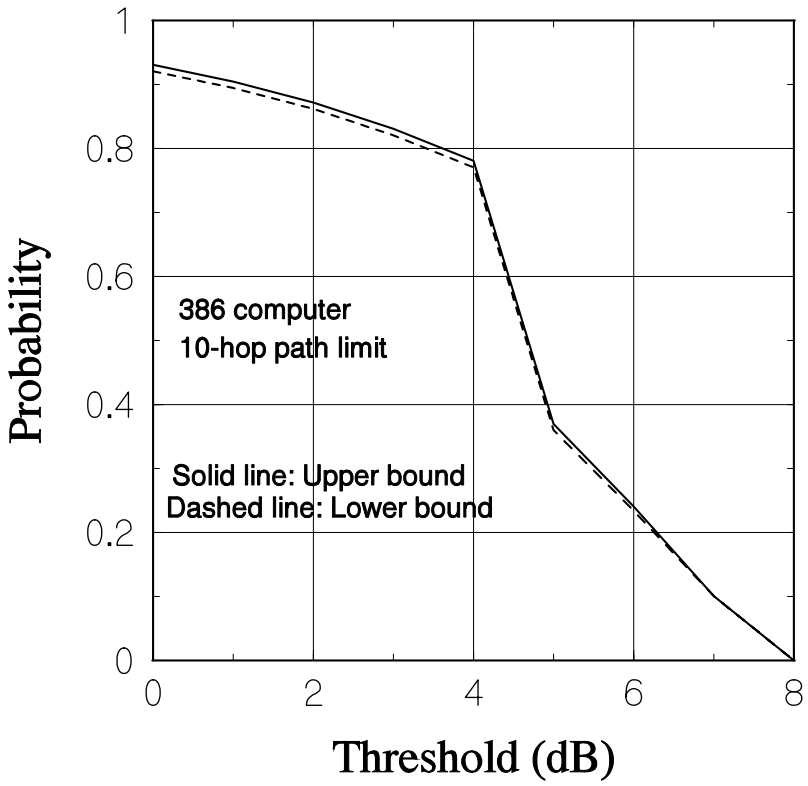


FIGURE 2-14 BOUNDS FOR 15-NODE EXAMPLE VS. THRESHOLD

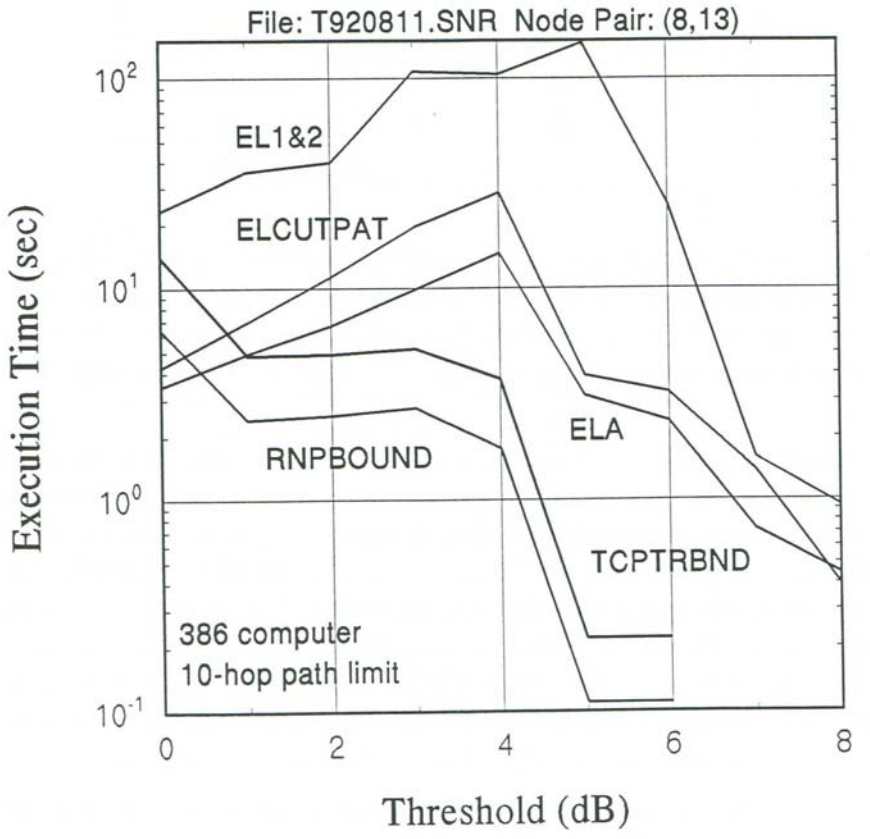


FIGURE 2-15 15-NODE EXECUTION TIME VS. THRESHOLD

TABLE 2-6 LINKS UP VS. THRESHOLD

Threshold	Links UP	Out of source	Into sink
0	60	2	6
1	55	2	4
2	53	2	4
3	51	2	4
4	48	2	4
5	41	1	4
6	36	1	4
7	31	1	4
8	25	(8, 13) disconnected	

This general behavior is different from that of the TCPTRBND and RNPBOUND programs, which experience an initial sharp decrease in the execution time as the threshold is raised to 1 dB, level off for thresholds between 1 and 3 dB, then decrease to a very small time as the threshold is increased further. This behavior is explained by the data in Table 2-6, which indicates that the number of links entering the sink node decreases by two as the threshold increases from 0 to 1 dB, and the number of links exiting the source node decreases by one as the threshold increases from 4 to 5 dB (as noted previously).

Figure 2-16 indicates the tightness of the upper and lower bounds for the programs. Not surprisingly, the equivalent-links programs generally maintain a difference of 0.01 between the bounds because of the program stopping criterion, but the relation between the bounds for the two recursive programs varies. The fact that the difference becomes zero for the factoring programs when the threshold is 5 dB or greater shows that for these conditions the exact probability is computed instead of the bounds, none of the links having reliabilities above the probability thresholds. A difference of zero is attributed to the EL1&2 program for thresholds of 6, 7, and 8 dB because the program truncated after recognizing that the lower bound was the exact probability.

The fact that the RNPBOUND program is both faster and more accurate (for this network example, at least) makes it a strong candidate for general use in calculating *s*-*t* reliabilities, in preference to the ELA. However, it will be shown below by example that the ELA becomes more competitive with RNPBOUND for larger networks.

2.3.2.3 Tests for the 34-node example network

Three sets of parametric algorithm performance results are presented below for the 34-node network, both for node pair (25, 20). The first set focuses on how the performance of the ELA changes when the EQLNKTST stopping criterion is varied; the second set makes a point about the effect of a limit on path lengths; and the third set compares the speed and accuracy of the several programs being considered.

Dependence of ELA performance upon the stopping criterion. The program EQLNKTST was exercised with a variation in the parameter ϵ, where the program will stop accumulating upper and lower bounds on the *s*-*t* reliability when $\mathrm{UB} - \mathrm{LB} < \epsilon$.

For reference, the bounds computed when $\epsilon = 0.01$ are shown in Figure 2-17. Note from this figure that there is very little change in the bounds until the threshold is raised

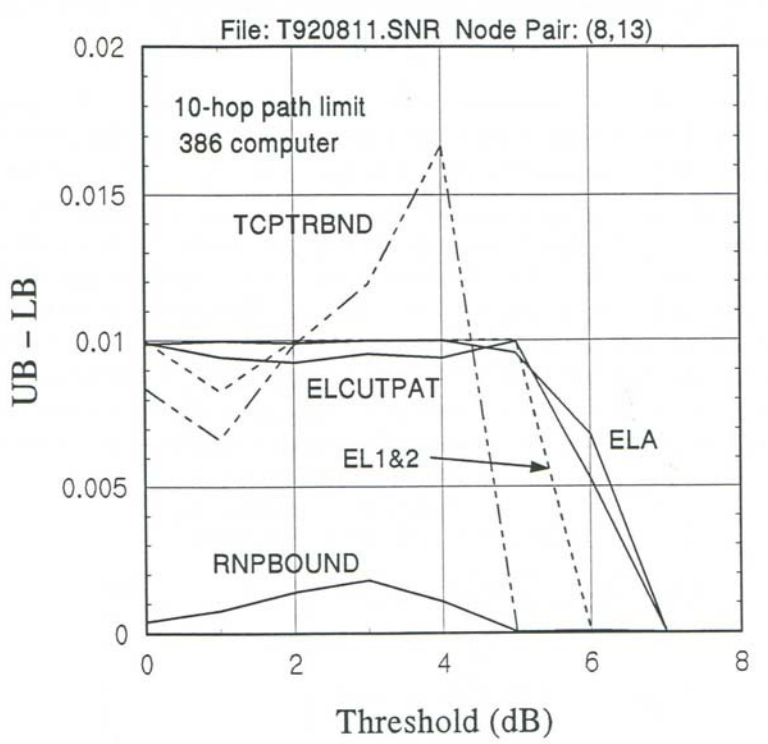


FIGURE 2-16 15-NODE BOUND TIGHTNESS VS. THRESHOLD

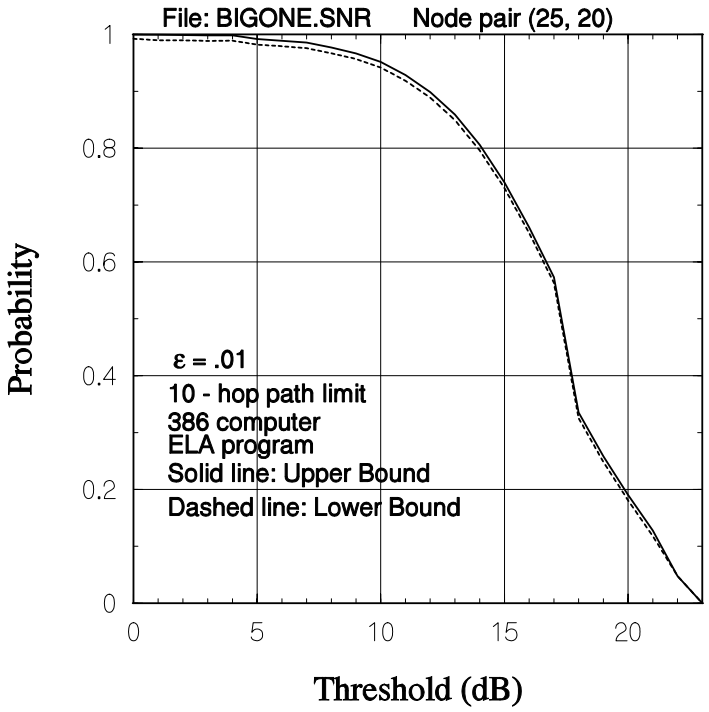


FIGURE 2-17 ELA BOUNDS FOR 34-NODE EXAMPLE VS. THRESHOLD

to 5 dB; as seen in Table 2-7 below, at this threshold value the number of UP links entering node 20, the sink, decreases from 2 to 1. The reason for the steep drop in reliability that is evidenced in Figure 2-17 when the threshold changes from 17 to 18 dB cannot be discerned from Table 2-7; however, upon detailed examination it was found that a key link (node 13 to node 8) goes DOWN when the threshold reaches 18 dB.

The execution time of the EQLNKTST implementation of the ELA is shown in Figure 2-19 as a function of the threshold for $\epsilon = 0.01$, 0.02, and 0.03. As expected, the time for the bounds to converge increases as the criterion for the convergence decreases, that is, as the bounds are forced to be tighter. The largest time to compute bounds for $\gamma_{25,20}$, 78.0 seconds, occurs for a 17 dB threshold and $\epsilon = 0.01$; this time is reduced by about 10% to 70.1 seconds if $\epsilon = 0.02$, or about 16% to 65.4 seconds if $\epsilon = 0.03$.

TABLE 2-7 LINKS UP VS. THRESHOLD

Threshold	Links UP	Out of source	Into sink
0	126	5	2
1	124	4	2
2-3	123	4	2
4	121	4	2
5-6	119	4	1
7-12	118	4	1
13	116	4	1
14-15	114	4	1
16	113	4	1
17	109	4	1
18	107	4	1
19	104	4	1
20	102	4	1
21	96	4	1
22	86	2	1
23	77	(25, 20) disconnected	

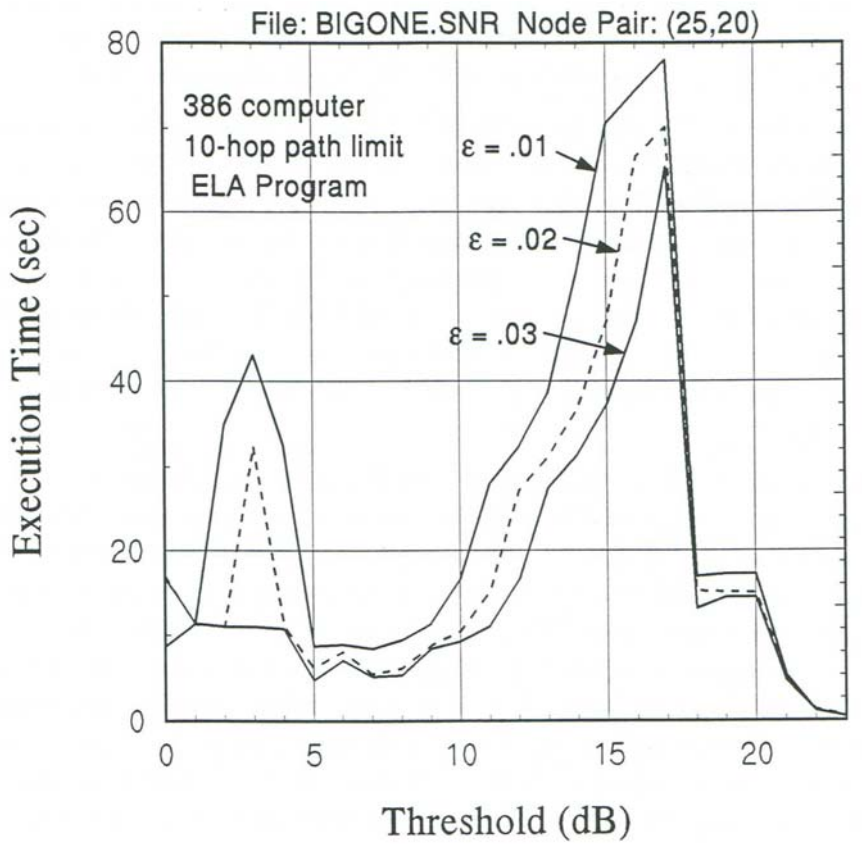


FIGURE 2-18 ELA EXECUTION TIME FOR 34-NODES VS. THRESHOLD

Note in Figure 2-18 that the tendency is for the execution time to grow exponentially as the threshold is increased, until some key link or combination of links by going DOWN simplifies the network topology. The growth in execution time is related directly to the number of success and failure events that are necessary to be processed, as shown by comparing Figure 2-18 with Figure 2-19. The explanation for the phenomena observed in these two figures is that, as the link SNRs deteriorate, the probabilities of the various events constituting the total success event or the total failure event become comparable in value, and it is necessary to include more of them to make the bounds tight.

For example, if all the links have reliability $\beta = 1.0$, then the partitioning finds only one success event (the shortest path) with probability 1.0 (assuming perfect nodes), and no failure events (all of which have zero probability); the other possible disjoint success events have zero probability because they are predicated on the failure of one or more links in the shortest path. If all the links have reliability just over $\beta = 0.5$, then each of

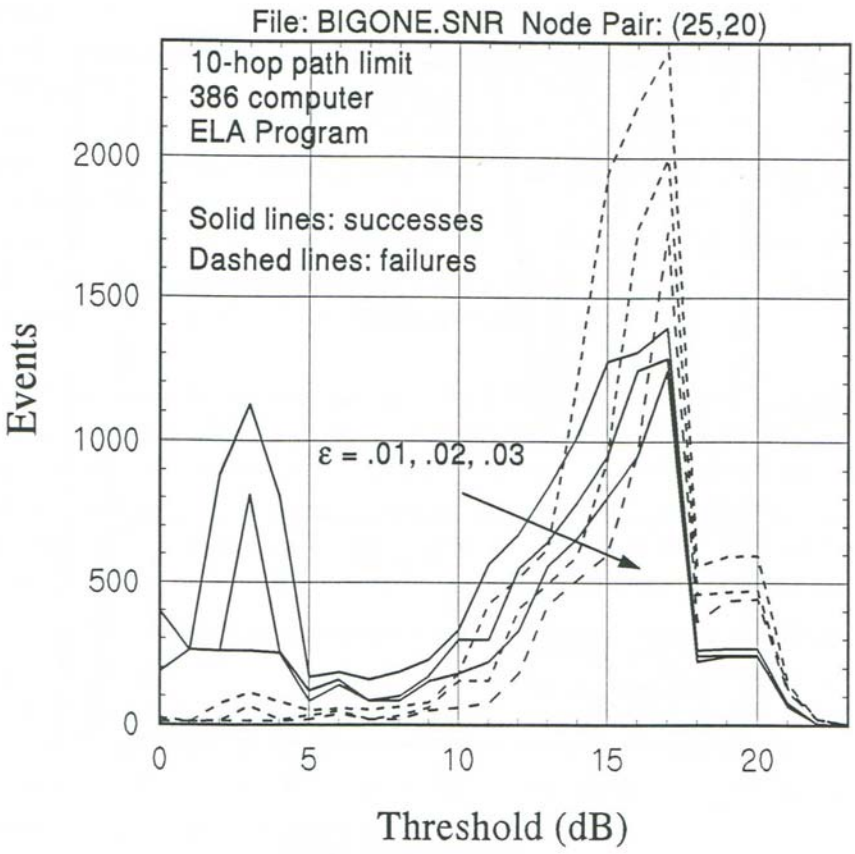


FIGURE 2-19 ELA EVENTS FOR 34-NODE EXAMPLE VS. THRESHOLD

the 2^M possible combinations of link successes and failures have the same probability, and the accumulation of the lower bound depends upon the distribution of path lengths—the more shorter paths there are, the larger the initial success events and the faster the lower bound will accumulate. However, as the threshold increases, causing various links to go DOWN, generally the longer, more circuitous paths survive.

The tightness of the bounds as a function of the threshold is depicted in Figure 2-20. Note in that figure that for higher values of the threshold the difference between the bounds tends to be very close to the value of ϵ, while for lower values of the threshold it is quite often the case that the bounds are closer together than ϵ. This behavior is due to the discreteness of the event probabilities. Initially both success and failure events contribute relatively large “chunks" of reliability to the accumulation of the bounds for low threshold values; the addition of a particular event's probability to the accumulation therefore is likely to “overshoot" the goal represented by the convergence criterion.

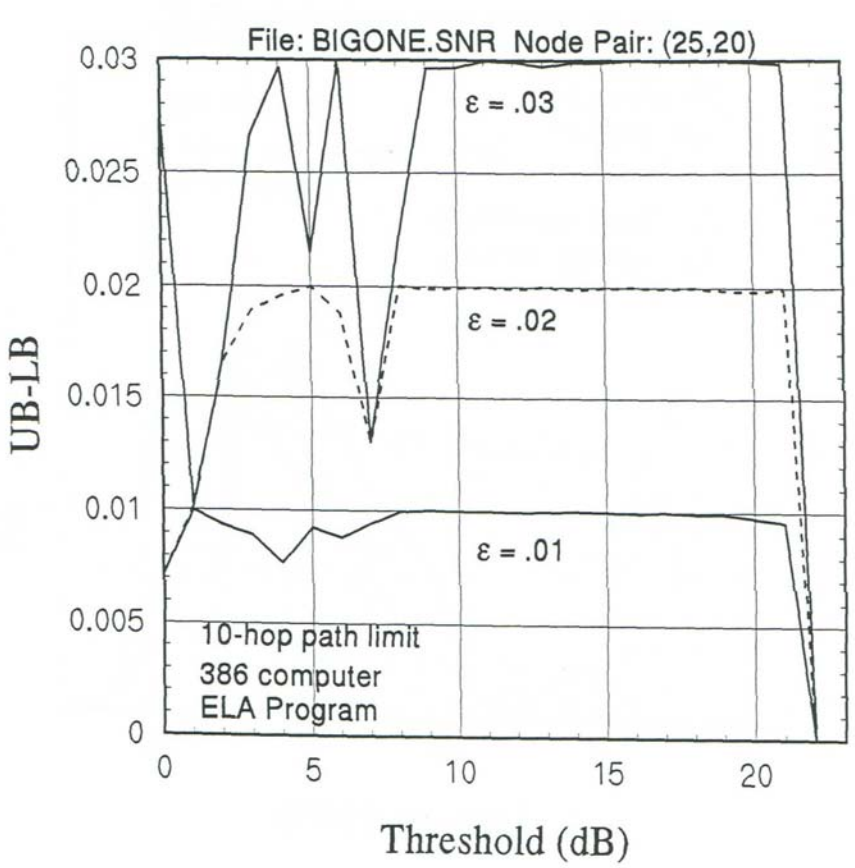


FIGURE 2-20 ELA BOUND TIGHTNESS FOR 34-NODES VS. THRESHOLD

Effect of path hop limit. In Figure 2-21, the bounds produced by EQLNKTST and RNPBOUND are compared, revealing an important fundamental difference between the algorithms that these programs implement. For SNR thresholds up to 7 dB, the two programs give virtually the same upper bound, with the RNPBOUND lower bound being tighter than that of EQLNKTST. However, for thresholds greater than 7 dB, RNPBOUND while maintaining close upper and lower bounds calculates a lower bound higher in value than the upper bound calculated by EQLNKTST. This result is startling at first, but has a very good explanation: the equivalent-links algorithm features a limit on the number of hops allowed in an $s \rightarrow t$ path, but there is no hop-limiting mechanism in the reduce and partition algorithm. Therefore, RNPBOUND calculates the s-t reliability unconstrained by path length, a quantity that is lower-bounded by the s-t reliability constrained by path length.

Algorithm performance comparisons. A comparison of program execution times is shown in Figure 2-22. For the three equivalent-links programs, the time begins to

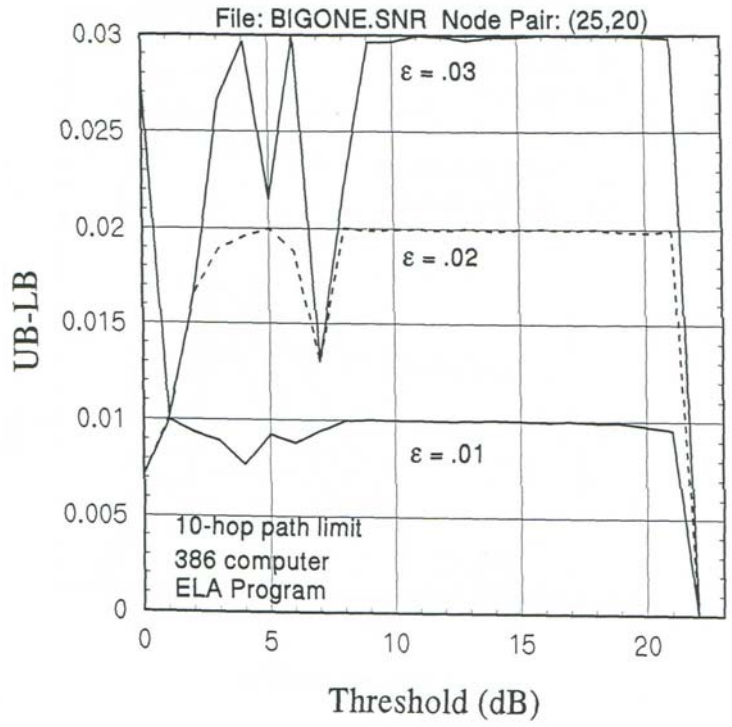


FIGURE 2-21 COMPARISON OF BOUNDS

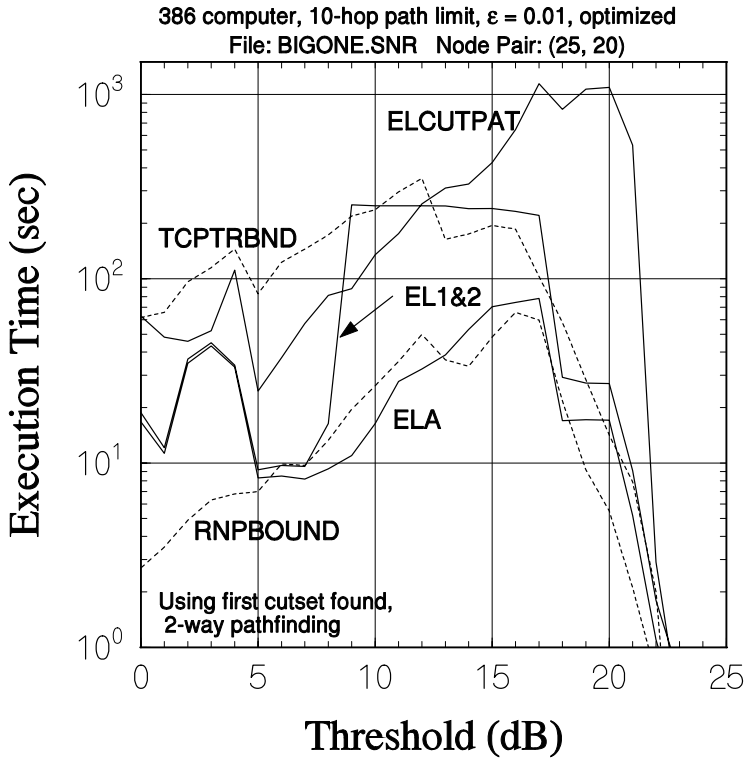


FIGURE 2-22 COMPARISON OF EXECUTION TIMES FOR 34 NODES

increase as the threshold rises, but drops in going from a threshold value of 4 dB to a value of 5 dB because, as Table 2-7 shows, the number of links going into the sink (node 20) then decreases from two to one. Thereafter the time for the ELA program again rises until the previously noted critical link failure occurs when the threshold is 18 dB.

The time for EL1&2 rises very sharply and remains nearly constant for threshold values of 9 to 17 dB; this behavior is due to the fact that for thresholds greater than 8 dB, the program runs until the ELA portion of the program computes a lower bound that is the exact reliability. Evidently, there are many cutsets with very small event probabilities, causing the upper bound that is computed by the ELA2 portion of the program to converge very slowly in this case.

The time for ELCUTPAT is about an order of magnitude greater than that for the ELA program for most of the threshold values, and rises steadily until it reaches about 1000 seconds, when the program is terminated due to the upper limit on the number of events, rather than due to bound convergence, as will be show below.

The times for the two programs based on factoring follow the same general trend as the time for the ELA program, though being an order of magnitude different for most of the range of threshold values shown.

In terms of speed, it appears from this example that the program implementing the ELA and the RNPBOUND program are the programs of choice for calculating s-t reliability for large networks, whereas for the smaller network example shown in Figure 2-15, both RNPBOUND and TCPTRBND are faster than the ELA program. The poor speed of ELCUTPAT and EL1&2, due evidently to the proliferation of small cutsets for this example, eliminate these programs from consideration.

The accuracies of the set of programs are shown in Figure 2-23, and confirm some of the statements made concerning Figure 2-22. For example, Figure 2-23 shows that the exact answer is computed by EL1&2 for thresholds greater than 8 dB, and that the ELCUTPAT bounds do not converge to the criterion $\mathrm{UB} - \mathrm{LB} < \epsilon$ for threshold values of 17, 19, and 20 dB. The accuracy of the ELA program is nearly constant at $\epsilon = 0.01$ until the network is so much simplified (at a threshold of 22 dB) that the exact reliability is computed.

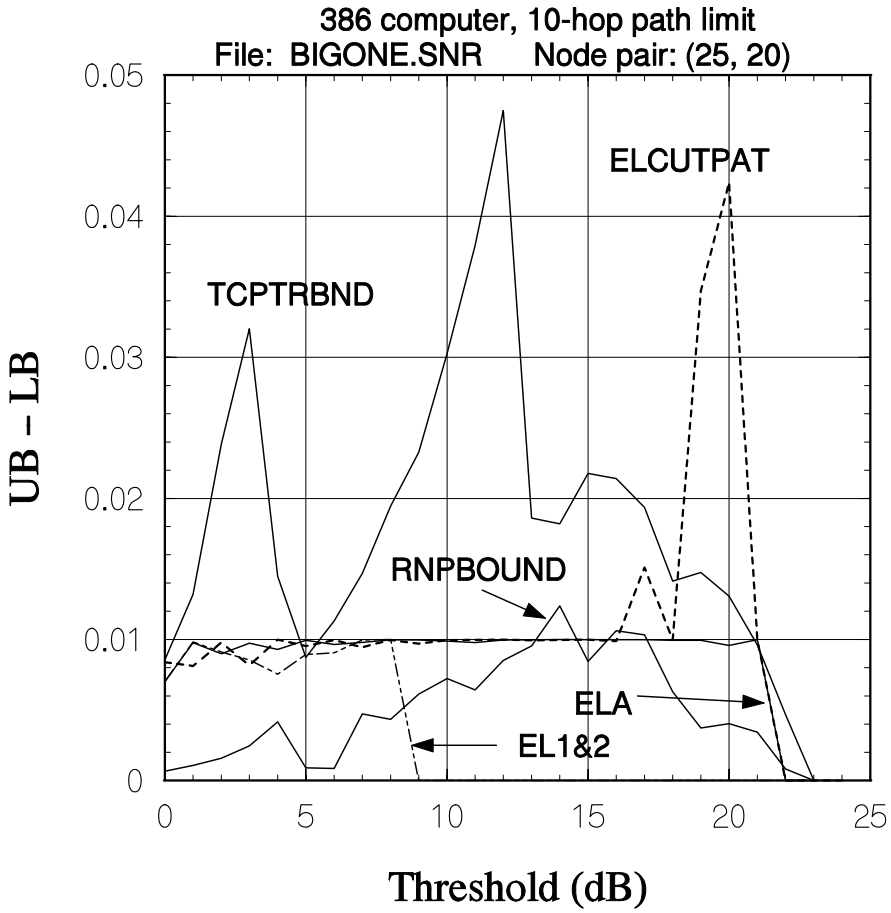


FIGURE 2-23 COMPARISON OF ACCURACIES FOR 34 NODES

The accuracies of TCPTRBND and RNPBOUND both fluctuate, since these programs do not terminate based on bound convergence. Evidently, the fact that RNPBOUND uses partioning in addition to factoring not only makes it run faster than TCPTRBND but also guarantees that it is more accurate for the same adaptive probability thresholding strategy; this trend was observable also for the 15-node example.

A check on the execution times for RNPBOUND and the ELA program for node pairs other than (25, 20) revealed a significant variety of worst-case times. The difficulty in calculating the *s*-*t* reliability for an arbitrary node pair is very dependent upon the particular network configuration and connectivity as a function of the SNR threshold. For some values of the threshold, sensitivity of the convergence of the bounds to the criterion ϵ is so great that the time for $\epsilon = 0.01$ is an order of magnitude greater than that for $\epsilon = 0.02$, for example. Therefore, some means of adapting the convergence criterion is desirable.

2.4 CONCLUSIONS AND RECOMMENDATIONS

On the basis of the numerical results produced by this study, the following was concluded in [13]:

- The concept of using cutsets instead of paths when calculating the upper bound on s-t reliability does work, but only for smaller networks or for large networks with relatively high link reliabilities. For larger networks, at some point, usually when the link reliabilities are not high, the events generated by processing cutset failure events become both numerous and quite small in probability, requiring excessive computational time.

- While a program combining cutsets and paths performed reasonably well, of the partitioning class of algorithms the original equivalent-links algorithm (with path-finding only) seems to work the best in terms of both speed and accuracy.

- A new algorithm combining partitioning and network reduction techniques, modified to include adaptive heuristic probability thresholds in order to calculate bounds, shows potential for calculating s-t reliabilities faster than the ELA.[8] However, this new approach does not include a convenient mechanism for stipulating a limit on the lengths of paths, nor one for trading off the prescribed accuracy with the run time, which are important considerations in modelling actual networks. It also, unlike the equivalent-links program, does not truncate on the basis of the tightness of the bounds, and cannot be used to store a file of events for later calculation.

On the basis of these conclusions and observations, the algorithm of choice is the equivalent-links algorithm.

If further studies are to be conducted in this area, it is recommended that attention be given to (1) further development of the equivalent-links algorithm, with emphasis on adapting the bound convergence criterion to render a better tradeoff between accuracy and speed, and (2) further development of the reduction and partition algorithm with adaptive probability thresholding, with emphasis on the adaptation technique and on methods of accounting for a path hop limit.

[8]The ELA implementation that was tested computed the success and failure probabilities using only link reliabilities; the *BackFit* procedure for including node reliabilities was not used. When node failures are taken into account by the ELA, it can be expected that its execution time will increase slightly. The RNPBOUND program that was tested did take into account node failures.

APPENDIX A

EXAMPLE OF A PROBABILITY CALCULATION FOR A 3 × 3 NETWORK USING THE EQUIVALENT-LINKS ALGORITHM (ORIGIN = 1, DESTINATION = 2)

As an example of the operation of the equivalent-links algorithm for including nodes in the s-t reliability calculation, we consider the problem of finding the probability that, in the 3×3 network of Figure A-1[9], the flood search succeeds in finding a path from node 1 to node 2. New events are assumed to be generated using path descriptions that give the links used in the order in which they are used.

A.1 FINDING THE SUCCESS AND FAILURE EVENTS

Initially, $\mathcal{W} = \{W_1 = [0\ \ 0\ \ 0 \cdots 0]\}$, the universal event. Using an unspecified pathfinding algorithm, we proceed as follows. The source is $s = 1$ and the terminal is $t = 2$. The first (shortest) completed path is found to be the set of elements

$$P = (10) \tag{A-1}$$

which includes, in this instance, the link which directly connects s and t. Since we have successfully found a path, we add $W_1 \cap P$ to $\mathcal{S}$:

$$S_1 = (10) \tag{A-2}$$

and add the complement, $W_1 \cap \overline{P}$ to $\mathcal{Y}$[10]:

$$Y_1 = (\overline{10}) \tag{A-3}$$

The initial working set $\mathcal{W}$ is now exhausted. Since $|\mathcal{Y}| = 1$, we are not done. We set $\mathcal{W} \leftarrow \mathcal{Y}$, i.e. set

$$W_1 = (\overline{10}) \tag{A-4}$$

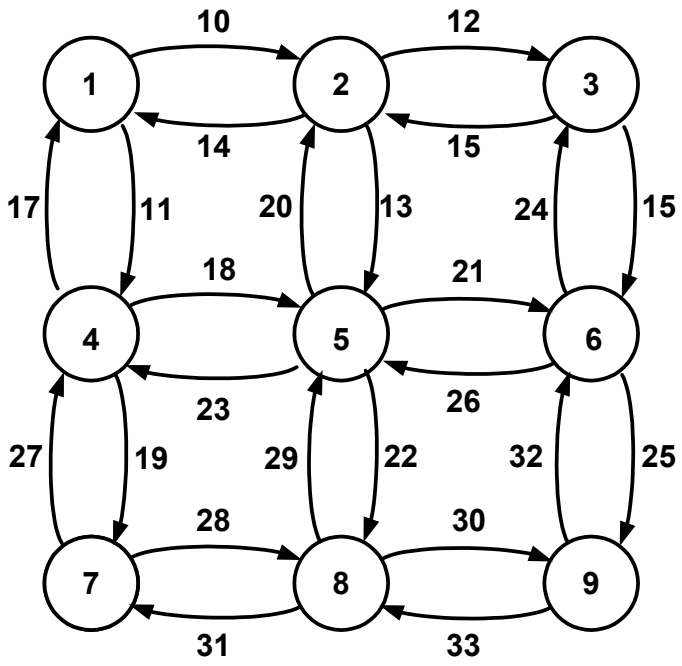
and then set

$$\mathcal{Y} = \emptyset. \tag{A-5}$$

On the second cycle through the algorithm, we have $|\mathcal{W}| = 1$ and therefore must

[9]The numbering of the links in Figure A-1 is slightly different than that in Figure 1-1 of the text.

[10] Only one first-in-first-out queue is required. For ease of presentation we speak of the collection $\mathcal{W}$ of events being read from the queue, and the collection $\mathcal{Y}$ of events being added to the queue.



Nodes:

$\Pr\{n_i\} = \alpha_i$

Links:

$\Pr\{l_k\} = \beta_k$

Alternate link notation:

$l_{10} \equiv l_{1,2}$ etc.

(a) Weighted Graph

$$\boldsymbol{G} = \begin{bmatrix} 0 & 10 & 0 & 11 & 0 & 0 & 0 & 0 & 0 \\ 14 & 0 & 12 & 0 & 13 & 0 & 0 & 0 & 0 \\ 0 & 16 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 17 & 0 & 0 & 0 & 18 & 0 & 19 & 0 & 0 \\ 0 & 20 & 0 & 23 & 0 & 21 & 0 & 22 & 0 \\ 0 & 0 & 24 & 0 & 26 & 0 & 0 & 0 & 25 \\ 0 & 0 & 0 & 27 & 0 & 0 & 0 & 28 & 0 \\ 0 & 0 & 0 & 0 & 29 & 0 & 31 & 0 & 30 \\ 0 & 0 & 0 & 0 & 0 & 32 & 0 & 33 & 0 \end{bmatrix}.$$

(b) Combined Node Adjacency and Link Identification Matrix

FIGURE A-1 EXAMPLE 3 × 3 NETWORK

examine one event. Now we examine W_1, seeking a short path from $s = 1$ to $t = 2$ and find the path using the node sequence 1, 4, 5, 2:

$$P = (11, 18, 20). \tag{A-6}$$

Note that in (A-6), the links have been listed in the order of the sequence of links used by the path that was found; in this instance, the order is the same as the arbitrary order in which the links were numbered. Since we have successfully found a path, we append $W_1 \cap P$ to $\mathcal{S}$,

$$S_2 = (\overline{10}, 11, 18, 20), \tag{A-7}$$

and add the partitioned form of the complement, $W_1 \cap \overline{P}$, to $\mathcal{Y}$:

$$Y_1 = (\overline{10}, \overline{11}) \tag{A-8a}$$

$$Y_2 = (\overline{10}, 11, \overline{18}) \tag{A-8b}$$

$$Y_3 = (\overline{10}, 11, 18, \overline{20}). \tag{A-8c}$$

We have now exhausted the working set $\mathcal{W}$. Since $|\mathcal{Y}| = 3 > 0$, we must perform another cycle through the algorithm. We set $\mathcal{W} \leftarrow \mathcal{Y}$, clear $\mathcal{Y} \leftarrow \emptyset$, and repeat the process. The new working set is obtained by setting $\mathcal{W}$ equal to the temporary collection $\mathcal{Y}$ as given in (A-8):

$$W_1 = (\overline{10}, \overline{11}) \tag{A-9a}$$

$$W_2 = (\overline{10}, 11, \overline{18}) \tag{A-9b}$$

$$W_3 = (\overline{10}, 11, 18, \overline{20}). \tag{A-9c}$$

and clearing $\mathcal{Y} = \emptyset$. We begin the next cycle through the algorithm by examining W_1. With the failure of both links 10 and 11, there is no operable link out of node 1, and a search for a short path fails quickly. W_1 is appended to the failure collection $\mathcal{F}$:

$$F_1 = (\overline{10}, \overline{11}) \tag{A-10}$$

We now examine W_2 of (A-9b). We seek a short path from 1 to 2, and find the path using the node sequence 1, 4, 7, 8, 5, 2:

$$P = (11, 19, 28, 29, 20). \tag{A-11}$$

Intersecting the path P of (A-11) with W_2 of (A-9b) gives a new member of the success collection:

$$S_3 = (\overline{10}, 11, \overline{18}, 19, 20, 28, 29). \tag{A-12}$$

Intersecting $\overline{P}$ with W_2 adds to the new temporary collection $\mathcal{Y}$

$$Y_1 = (\overline{10}, 11, \overline{18}, \overline{19}) \tag{A-13a}$$

$$Y_2 = (\overline{10}, 11, \overline{18}, 19, \overline{28}) \tag{A-13b}$$

$$Y_3 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}) \tag{A-13c}$$

$$Y_4 = (\overline{10}, 11, \overline{18}, 19, \overline{20}, 28, 29) \tag{A-13d}$$

We now examine W_3 given by (A-9c). The path search finds the ordered path using the node sequence 1, 4, 5, 6, 3, 2:

$$P = (11, 18, 21, 24, 16). \tag{A-14}$$

We intersect P given by (A-14) with W_3 given by (A-9c) to obtain

$$S_4 = (\overline{10}, 11, 16, 18, \overline{20}, 21, 24) \tag{A-15}$$

and append the partitioned complement of $\overline{P}$ intersected with W_3 to the temporary collection $\mathcal{Y}$:

$$Y_5 = (\overline{10}, 11, 18, \overline{20}, \overline{21}) \tag{A-16a}$$

$$Y_6 = (\overline{10}, 11, 18, \overline{20}, 21, \overline{24}) \tag{A-16b}$$

$$Y_7 = (\overline{10}, 11, \overline{16}, 18, \overline{20}, 21, 24). \tag{A-16c}$$

This exhausts the working collection (A-9). Since $|\mathcal{Y}| > 0$, we must perform another pass through the algorithm with the new working set given by the set of terms in (A-13) and (A-16).

We examine W_1 given by (A-13a). The pathfinding routine returns failure and we append W_1 to the failure collection $\mathcal{F}$:

$$F_2 = (\overline{10}, 11, \overline{18}, \overline{19}). \tag{A-17}$$

Examination of W_2 given by (A-13b) also results in failure and W_2 is added to $\mathcal{F}$:

$$F_3 = (\overline{10}, 11, \overline{18}, 19, \overline{28}) \tag{A-18}$$

We then examine W_3 given by (A-13c). The search results in finding the path using the node sequence 1, 4, 7, 8, 9, 6, 3, 2:

$$P = (11, 19, 28, 30, 32, 24, 16). \tag{A-19}$$

We append the intersection of W_3 and the path P to the success collection $\mathcal{S}$

$$S_5 = (\overline{10}, 11, 16, \overline{18}, 19, 24, 28, \overline{29}, 30, 32), \tag{A-20}$$

and we append the partitioned complement, $W_3 \cap \overline{P}$, to the temporary collection $\mathcal{Y}$:

$$Y_1 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, \overline{30}) \tag{A-21a}$$

$$Y_2 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 28,\ \overline{29},\ 30,\ \overline{32}) \tag{A-21b}$$

$$Y_3 = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{24},\ 28,\ \overline{29},\ 30,\ 32) \tag{A-21c}$$

$$Y_4 = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 24,\ 28,\ \overline{29},\ 30,\ 32) \tag{A-21d}$$

We examine W_4 given by (A-13d). The search finds the path using the node sequence 1, 4, 7, 8, 5, 6, 3, 2:

$$P = (11,\ 19,\ 28,\ 29,\ 21,\ 24,\ 16). \tag{A-22}$$

We append the intersection of W_4 with the path P to the success collection $\mathcal{S}$:

$$S_6 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ 21,\ 24,\ 28,\ 29) \tag{A-23}$$

and append the partitioned complement $W_4 \cap \overline{P}$ to the temporary collection $\mathcal{Y}$:

$$Y_5 = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 28,\ 29) \tag{A-24a}$$

$$Y_6 = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{20},\ 21,\ \overline{24},\ 28,\ 29) \tag{A-24b}$$

$$Y_7 = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ \overline{20},\ 21,\ 24,\ 28,\ 29). \tag{A-24c}$$

We examine W_5 given by (A-16a). The search finds the path using the node sequence 1, 4, 5, 8, 9, 6, 3, 2:

$$P = (11,\ 18,\ 22,\ 30,\ 32,\ 24,\ 16). \tag{A-25}$$

We append the intersection of W_5 with the path P to the success collection $\mathcal{S}$:

$$S_7 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ \overline{21},\ 22,\ 24,\ 30,\ 32) \tag{A-26}$$

and append the partitioned complement $W_5 \cap \overline{P}$ to the temporary collection $\mathcal{Y}$:

$$Y_8 = (\overline{10},\ 11,\ 18,\ \overline{20},\ \overline{21},\ \overline{22}) \tag{A-27a}$$

$$Y_9 = (\overline{10},\ 11,\ 18,\ \overline{20},\ \overline{21},\ 22,\ \overline{30}) \tag{A-27b}$$

$$Y_{10} = (\overline{10},\ 11,\ 18,\ \overline{20},\ \overline{21},\ 22,\ 30,\ \overline{32}) \tag{A-27c}$$

$$Y_{11} = (\overline{10},\ 11,\ 18,\ \overline{20},\ \overline{21},\ 22,\ \overline{24},\ 30,\ 32) \tag{A-27d}$$

$$Y_{12} = (\overline{10},\ 11,\ \overline{16},\ 18,\ \overline{20},\ \overline{21},\ 22,\ 24,\ 30,\ 32). \tag{A-27e}$$

We find that W_6 and W_7 given by (A-16b) and (A-16c), respectively, fail to find a path reaching the destination. We append W_6 and W_7 to the failure collection $\mathcal{F}$:

$$F_4 = (\overline{10},\ 11,\ 18,\ \overline{20},\ 21,\ \overline{24}) \tag{A-28a}$$

$$F_5 = (\overline{10},\ 11,\ \overline{16},\ 18,\ \overline{20},\ 21,\ 24). \tag{A-28b}$$

This exhausts the working set $\mathcal{W}$ given by (A-13) and (A-16). The temporary collection $\mathcal{Y}$ given by (A-21), (A-24), and (A-27) has cardinality greater than zero. Therefore we must make another pass through the algorithm with the new working collection given by (A-21), (A-24), and (A-27) and $\mathcal{Y} \leftarrow \emptyset$.

We first examine W_1 given by (A-21a). The search fails to find a path to the destination and we append W_1 to $\mathcal{F}$:

$$F_6 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, \overline{30}). \tag{A-29}$$

Next we examine W_2 given by (A-21b). The search fails to find a path to the destination and we append W_2 to $\mathcal{F}$:

$$F_7 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, 30, \overline{32}). \tag{A-30}$$

We examine W_3 given by (A-21c). The path-finding algorithm finds the path using the node sequence 1, 4, 7, 8, 9, 6, 5, 2:

$$P = (11, 19, 28, 30, 32, 26, 20). \tag{A-31}$$

We append the intersection of W_3 with P to the success collection $\mathcal{S}$:

$$S_8 = (\overline{10}, 11, \overline{18}, 19, 20, \overline{24}, 26, 28, \overline{29}, 30, 32) \tag{A-32}$$

and append the partitioned form of the intersection $W_3 \cap \overline{P}$ to $\mathcal{Y}$:

$$Y_1 = (\overline{10}, 11, \overline{18}, 19, \overline{24}, \overline{26}, 28, \overline{29}, 30, 32) \tag{A-33a}$$

$$Y_2 = (\overline{10}, 11, \overline{18}, 19, \overline{20}, \overline{24}, 26, 28, \overline{29}, 30, 32). \tag{A-33b}$$

We examine W_4 given by (A-21d). The pathfinding algorithm finds the path using the node sequence 1, 4, 7, 8, 9, 6, 5, 2:

$$P = (11, 19, 28, 30, 32, 26, 20). \tag{A-34}$$

We append the intersection of W_4 and the path P to the success collection $\mathcal{S}$:

$$S_9 = (\overline{10}, 11, \overline{16}, \overline{18}, 19, 20, 24, 26, 28, \overline{29}, 30, 32) \tag{A-35}$$

and we append the partitioned form of $W_4 \cap \overline{P}$ to the temporary collection $\mathcal{Y}$:

$$Y_3 = (\overline{10}, 11, \overline{16}, \overline{18}, 19, 24, \overline{26}, 28, \overline{29}, 30, 32) \tag{A-36a}$$

$$Y_4 = (\overline{10}, 11, \overline{16}, \overline{18}, 19, \overline{20}, 24, 26, 28, \overline{29}, 30, 32) \tag{A-36b}$$

We examine W_5 given by (A-24a). The pathfinding algorithm finds the path using the node sequence 1, 4, 7, 8, 9, 6, 3, 2:

$$P = (11, 19, 28, 30, 32, 24, 16). \tag{A-37}$$

We take the intersection of W_5 with the path P and append the intersection to the success collection $\mathcal{S}$:

$$S_{10} = (\overline{10}, 11, 16, \overline{18}, 19, \overline{20}, \overline{21}, 24, 28, 29, 30, 32) \tag{A-38}$$

and append the partitioned form of $W_5 \cap \overline{P}$ to the temporary collection $\mathcal{Y}$:

$$Y_5 = (\overline{10}, 11, \overline{18}, 19, \overline{20}, \overline{21}, 28, 29, \overline{30}) \tag{A-39a}$$

$$Y_6 = (\overline{10}, 11, \overline{18}, 19, \overline{20}, \overline{21}, 28, 29, 30, \overline{32}) \tag{A-39b}$$

$$Y_7 = (\overline{10}, 11, \overline{18}, 19, \overline{20}, \overline{21}, \overline{24}, 28, 29, 30, 32) \tag{A-39c}$$

$$Y_8 = (\overline{10}, 11, \overline{16}, \overline{18}, 19, \overline{20}, \overline{21}, 24, 28, 29, 30, 32). \tag{A-39d}$$

We examine W_6 given by (A-24b). The pathfinding algorithm does not find a path, so W_6 is added to the failure collection $\mathcal{F}$:

$$F_8 = (\overline{10}, 11, \overline{18}, 19, \overline{20}, 21, \overline{24}, 28, 29). \tag{A-40}$$

We then examine W_7 given by (A-24c). The pathfinding algorithm does not find a path, so we add W_7 to $\mathcal{F}$:

$$F_9 = (\overline{10}, 11, \overline{16}, \overline{18}, 19, \overline{20}, 21, 24, 28, 29). \tag{A-41}$$

Examination of W_8 given by (A-27a) results in finding the path using the node sequence 1, 4, 7, 8, 9, 6, 3, 2:

$$P = (11, 19, 28, 30, 32, 24, 16). \tag{A-42}$$

We form the intersection of W_8 and the path P and append that intersection to the success collection $\mathcal{S}$:

$$S_{11} = (\overline{10}, 11, 16, 18, 19, \overline{20}, \overline{21}, \overline{22}, 24, 28, 30, 32). \tag{A-43}$$

Also, we append the intersection $W_8 \cap \overline{P}$, in partitioned form, to the temporary collection $\mathcal{Y}$:

$$Y_9 = (\overline{10}, 11, 18, \overline{19}, \overline{20}, \overline{21}, \overline{22}) \tag{A-44a}$$

$$Y_{10} = (\overline{10}, 11, 18, 19, \overline{20}, \overline{21}, \overline{22}, \overline{28}) \tag{A-44b}$$

$$Y_{11} = (\overline{10}, 11, 18, 19, \overline{20}, \overline{21}, \overline{22}, 28, \overline{30}) \tag{A-44c}$$

$$Y_{12} = (\overline{10}, 11, 18, 19, \overline{20}, \overline{21}, \overline{22}, 28, 30, \overline{32}) \tag{A-44d}$$

$$Y_{13} = (\overline{10}, 11, 18, 19, \overline{20}, \overline{21}, \overline{22}, \overline{24}, 28, 30, 32) \tag{A-44e}$$

$$Y_{14} = (\overline{10}, 11, \overline{16}, 18, 19, \overline{20}, \overline{21}, \overline{22}, 24, 28, 30, 32). \tag{A-44f}$$

We examine W_9 through W_{12} given by (A-27b) to (A-27e). The search in each case fails to find a path to the destination, so we append W_9 through W_{11} to the failure collection $\mathcal{F}$:

$$F_{10} = (\overline{10},\, 11,\, 18,\, \overline{20},\, \overline{21},\, 22,\, \overline{30}) \tag{A-45a}$$

$$F_{11} = (\overline{10},\, 11,\, 18,\, \overline{20},\, \overline{21},\, 22,\, 30,\, \overline{32}) \tag{A-45b}$$

$$F_{12} = (\overline{10},\, 11,\, 18,\, \overline{20},\, \overline{21},\, 22,\, \overline{24},\, 30,\, 32) \tag{A-45c}$$

$$F_{13} = (\overline{10},\, 11,\, \overline{16},\, 18,\, \overline{20},\, \overline{21},\, 22,\, 24,\, 30,\, 32). \tag{A-45d}$$

This exhausts the working set $\mathcal{W}$ given by (A-21), (A-24), and (A-27). The temporary collection $\mathcal{Y}$ given by (A-33), (A-36), (A-39), and (A-44) is not empty. Therefore we must make another pass through the algorithm with the new working collection $\mathcal{W}$ given by $\mathcal{W} \leftarrow \mathcal{Y}$, then set $\mathcal{Y} \leftarrow \emptyset$.

Each of the Ws given by (A-33), (A-36), (A-39), and (A-44) results in the failure of the pathfinding algorithm to find a path from node 1 to node 2. Thus to the failure collection we add

$$F_{14} = (\overline{10},\, 11,\, \overline{18},\, 19,\, \overline{24},\, \overline{26},\, 28,\, \overline{29},\, 30,\, 32) \tag{A-46a}$$

$$F_{15} = (\overline{10},\, 11,\, \overline{18},\, 19,\, \overline{20},\, \overline{24},\, 26,\, 28,\, \overline{29},\, 30,\, 32) \tag{A-46b}$$

$$F_{16} = (\overline{10},\, 11,\, \overline{16},\, \overline{18},\, 19,\, 24,\, \overline{26},\, 28,\, \overline{29},\, 30,\, 32) \tag{A-46c}$$

$$F_{17} = (\overline{10},\, 11,\, \overline{16},\, \overline{18},\, 19,\, \overline{20},\, 24,\, 26,\, 28,\, \overline{29},\, 30,\, 32) \tag{A-46d}$$

$$F_{18} = (\overline{10},\, 11,\, \overline{18},\, 19,\, \overline{20},\, \overline{21},\, 28,\, 29,\, \overline{30}) \tag{A-46e}$$

$$F_{19} = (\overline{10},\, 11,\, \overline{18},\, 19,\, \overline{20},\, \overline{21},\, 28,\, 29,\, 30,\, \overline{32}) \tag{A-46f}$$

$$F_{20} = (\overline{10},\, 11,\, \overline{18},\, 19,\, \overline{20},\, \overline{21},\, \overline{24},\, 28,\, 29,\, 30,\, 32) \tag{A-46g}$$

$$F_{21} = (\overline{10},\, 11,\, \overline{16},\, \overline{18},\, 19,\, \overline{20},\, \overline{21},\, 24,\, 28,\, 29,\, 30,\, 32) \tag{A-46h}$$

$$F_{22} = (\overline{10},\, 11,\, 18,\, \overline{19},\, \overline{20},\, \overline{21},\, \overline{22}) \tag{A-46i}$$

$$F_{23} = (\overline{10},\, 11,\, 18,\, 19,\, \overline{20},\, \overline{21},\, \overline{22},\, \overline{28}) \tag{A-46j}$$

$$F_{24} = (\overline{10},\, 11,\, 18,\, 19,\, \overline{20},\, \overline{21},\, \overline{22},\, 28,\, \overline{30}) \tag{A-46k}$$

$$F_{25} = (\overline{10},\, 11,\, 18,\, 19,\, \overline{20},\, \overline{21},\, \overline{22},\, 28,\, 30,\, \overline{32}) \tag{A-46l}$$

$$F_{26} = (\overline{10},\, 11,\, 18,\, 19,\, \overline{20},\, \overline{21},\, \overline{22},\, \overline{24},\, 28,\, 30,\, 32) \tag{A-46m}$$

$$F_{27} = (\overline{10},\, 11,\, \overline{16},\, 18,\, 19,\, \overline{20},\, \overline{21},\, \overline{22},\, 24,\, 28,\, 30,\, 32). \tag{A-46n}$$

This exhausts the working set $\mathcal{W}$. The temporary collection $\mathcal{Y} = \emptyset$, so we are done! The collection $\mathcal{S}$ is now an exhaustive success collection and the collection $\mathcal{F}$ is an exhaustive failure collection. The probability of successfully routing a call from source 1 to destination 2 (s-t reliability with s = 1 and t = 2) may be computed using either $\mathcal{S}$ or $\mathcal{F}$.

A.2 ACCOUNTING FOR THE NODE RELIABILITIES

According the formulas given in the text as (1-14), the evaluation of the s-t reliability for this example, including node reliabilities, is accomplished by summing the following success probabilities:

$$\Pr\{S_1\} = \alpha_1\,[\alpha_2\beta_{10}] \tag{A-47a}$$

$$\Pr\{S_2\} = \alpha_1\,[\alpha_2(1-\beta_{10})\beta_{20}][\alpha_4\beta_{11}][\alpha_5\beta_{18}] \tag{A-47b}$$

$$\Pr\{S_3\} = \alpha_1\,[\alpha_2(1-\beta_{10})\beta_{20}][\alpha_4\beta_{11}][\alpha_5(1-\beta_{18})\beta_{29}][\alpha_7\beta_{19}][\alpha_8\beta_{28}] \tag{A-47c}$$

$$\Pr\{S_4\} = \alpha_1\,[\alpha_2(1-\beta_{10})\beta_{16}(1-\beta_{20})][\alpha_3\beta_{24}][\alpha_4\beta_{11}][\alpha_5\beta_{18}][\alpha_6\beta_{21}] \tag{A-47d}$$

$$\begin{aligned}\Pr\{S_5\} = \alpha_1\,&[\alpha_2(1-\beta_{10})\beta_{16}][\alpha_3\beta_{24}][\alpha_4\beta_{11}][\alpha_5(1-\beta_{18})(1-\beta_{29})+1-\alpha_5]\\ &\times[\alpha_6\beta_{32}][\alpha_7\beta_{19}][\alpha_8\beta_{28}][\alpha_9\beta_{30}]\end{aligned} \tag{A-47e}$$

$$\begin{aligned}\Pr\{S_6\} = \alpha_1\,&[\alpha_2(1-\beta_{10})\beta_{16}(1-\beta_{20})][\alpha_3\beta_{24}][\alpha_4\beta_{11}][\alpha_5(1-\beta_{18})\beta_{29}]\\ &\times[\alpha_6\beta_{21}][\alpha_7\beta_{19}][\alpha_8\beta_{28}]\end{aligned} \tag{A-47f}$$

$$\begin{aligned}\Pr\{S_7\} = \alpha_1\,&[\alpha_2(1-\beta_{10})\beta_{16}(1-\beta_{20})][\alpha_3\beta_{24}][\alpha_4\beta_{11}][\alpha_5\beta_{18}][\alpha_6\beta_{32}(1-\beta_{21})]\\ &\times[\alpha_8\beta_{22}][\alpha_9\beta_{30}]\end{aligned} \tag{A-47g}$$

$$\begin{aligned}\Pr\{S_8\} = \alpha_1[\alpha_2(1-\beta_{10})\beta_{20})][\alpha_3(1-\beta_{24})+1-\alpha_3][\alpha_4\beta_{11}][\alpha_5(1-\beta_{18})\beta_{26}(1-\beta_{29})]\\ \times[\alpha_6\beta_{32}][\alpha_7\beta_{19}][\alpha_8\beta_{28}][\alpha_9\beta_{30}]\end{aligned} \tag{A-47h}$$

$$\begin{aligned}\Pr\{S_9\} = \alpha_1\,&[\alpha_2(1-\beta_{10})(1-\beta_{16})\beta_{20}][\alpha_3\beta_{24}][\alpha_4\beta_{11}][\alpha_5(1-\beta_{18})\beta_{26}(1-\beta_{29})]\\ &\times[\alpha_6\beta_{32}][\alpha_7\beta_{19}][\alpha_8\beta_{28}][\alpha_9\beta_{30}]\end{aligned} \tag{A-47i}$$

$$\begin{aligned}\Pr\{S_{10}\} = \alpha_1\,&[\alpha_2(1-\beta_{10})\beta_{16}(1-\beta_{20})][\alpha_3\beta_{24}][\alpha_4\beta_{11}][\alpha_5(1-\beta_{18})\beta_{29}]\\ &\times[\alpha_6(1-\beta_{21})\beta_{32}][\alpha_7\beta_{19}][\alpha_8\beta_{28}][\alpha_9\beta_{30}]\end{aligned} \tag{A-47j}$$

$$\begin{aligned}\Pr\{S_{11}\} = \alpha_1\,&[\alpha_2(1-\beta_{10})\beta_{16}(1-\beta_{20})][\alpha_3\beta_{24}][\alpha_4\beta_{11}][\alpha_5\beta_{18}][\alpha_6(1-\beta_{21})\beta_{32}]\\ &\times[\alpha_7\beta_{19}][\alpha_8(1-\beta_{22})\beta_{28}][\alpha_9\beta_{30}]\end{aligned} \tag{A-47k}$$

APPENDIX B

EXAMPLE OF A PROBABILITY CALCULATION FOR A 3 × 3 NETWORK USING THE CUTSET METHOD (ORIGIN = 1, DESTINATION = 2)

As an example of the operation of the cutset method for the s-t reliability calculation, we consider the problem of finding the probability that, in the 3 × 3 network of Figure A-1, the flood search succeeds in finding a path from node 1 to node 2. This is the same example worked in Appendix A for the equivalent-links algorithm.

We begin with $\mathcal{W} = \{W_1 = [0\ \ 0\ \ 0 \cdots 0]\}$, the universal event. We seek a small cutset of DOWN links that precludes a path from node 1 to node 2. Using an unspecified minimum cutset algorithm that works both forward from the source and backward from the sink—and giving preferences to a cutset found in a forward search in case of a tie in the size of the cutsets found—we proceed as follows. The source is $s = 1$ and the terminal is $t = 2$. In the forward direction the smallest cutset is $(\overline{10}, \overline{11})$; that is, there is no path out from the source if links 10 and 11 are DOWN. In the reverse direction the smallest cutset is $(\overline{10}, \overline{16}, \overline{20})$; that is, there is no path into the sink if links 10, 16, and 20 are DOWN. Thus the selection is

$$C = (\overline{10}, \overline{11}) \tag{B-1}$$

which includes, in this instance, the link which directly connects s and t. Since we have successfully found a cutset, we add $W_1 \cap C$ to $\mathcal{F}$:

$$F_1 = (\overline{10}, \overline{11}) \tag{B-2}$$

and add the complement, $W_1 \cap \overline{C}$ to $\mathcal{Y}$[11]:

$$Y_1 = (10) \tag{B-3a}$$

$$Y_2 = (\overline{10}, 11). \tag{B-3b}$$

The initial working set $\mathcal{W}$ is now exhausted. Since $|\mathcal{Y}| = 2$, we are not done. We set $\mathcal{W} \leftarrow \mathcal{Y}$, i.e. set

$$W_1 = (10) \tag{B-4a}$$

$$W_2 = (\overline{10}, 11) \tag{B-4b}$$

and then set

[11]Only one first-in-first-out queue is required. For ease of presentation we speak of the collection $\mathcal{W}$ of events being read from the queue, and the collection $\mathcal{Y}$ of events being added to the queue.

$$\mathcal{Y} = \emptyset. \tag{B-5}$$

On the second cycle through the algorithm, we have $|\mathcal{W}| = 2$ and therefore must examine two events. Now we examine W_1, seeking a short cutset to preclude a path from $s = 1$ to $t = 2$ and find that there is none, since link 10 directly connects s to t and is postulated to be UP. Thus we have found

$$S_1 = (10), \tag{B-6}$$

and no further processing of W_1 is required.

With $W_2 = (\overline{10}, 11)$ postulated as the status of the network, the forward cutset is $(\overline{18}, \overline{19})$ and the reverse cutset is $(\overline{16}, \overline{20})$. Giving preference to the former, we have found a failure event, and append $W_2 \cap C$ to $\mathcal{F}$:

$$F_2 = (\overline{10}, 11, \overline{18}, \overline{19}), \tag{B-7}$$

The partitioned form of the complement, $W_2 \cap \overline{C}$, is added to $\mathcal{Y}$:

$$Y_1 = (\overline{10}, 11, 18) \tag{B-8a}$$

$$Y_2 = (\overline{10}, 11, \overline{18}, 19). \tag{B-8b}$$

We have now exhausted the working set $\mathcal{W}$. Since $|\mathcal{Y}| = 2 > 0$, we must perform another cycle through the algorithm. We set $\mathcal{W} \leftarrow \mathcal{Y}$, clear $\mathcal{Y} \leftarrow \emptyset$, and repeat the process. The new working set is obtained by setting $\mathcal{W}$ equal to the temporary collection $\mathcal{Y}$ as given in (B-8):

$$W_1 = (\overline{10}, 11, 18) \tag{B-9a}$$

$$W_2 = (\overline{10}, 11, \overline{18}, 19). \tag{B-9b}$$

and clearing $\mathcal{Y} = \emptyset$. We begin the next cycle through the algorithm by examining W_1. In the forward direction, links 19, 20, 21, and 22 must simultaneously be DOWN to preclude a path from node 1 to node 2; in the reverse direction, links 16 and 20 must be DOWN to preclude a path into node 2 from node 1. Therefore, the selected cutset is $(\overline{16}, \overline{20})$, giving rise to

$$F_3 = (\overline{10}, 11, \overline{16}, 18, \overline{20}) \tag{B-10}$$

and

$$Y_1 = (\overline{10}, 11, 16, 18) \tag{B-11a}$$

$$Y_2 = (\overline{10}, 11, \overline{16}, 18, 20). \tag{B-11b}$$

We now examine W_2 given by (B-9b). The forward cutset involves links only one link, link 28, and therefore is the preferred cutset, giving rise to

$$F_4 = (\overline{10}, 11, \overline{18}, 19, \overline{28}) \tag{B-12}$$

and

$$Y_3 = (\overline{10}, 11, \overline{18}, 19, 28). \tag{B-13}$$

This exhausts the working collection (B-9). Since $|\mathcal{Y}| > 0$, we must perform another pass through the algorithm with the new working set given by the set of events in (B-11) and (B-13).

We examine W_1 given by (B-11a). In the forward direction, links 19, 20, 21, and 22 must be DOWN to preclude a path; working backwards from the sink we find that links 20 and 24 must be DOWN. Therefore the selected cutset is $(\overline{20}, \overline{24})$, giving rise to

$$F_5 = (\overline{10}, 11, 16, 18, \overline{20}, \overline{24}) \tag{B-14}$$

and

$$Y_1 = (\overline{10}, 11, 16, 18, 20) \tag{B-15a}$$

$$Y_2 = (\overline{10}, 11, 16, 18, \overline{20}, 24). \tag{B-15b}$$

We then examine W_2 given by (B-11b). There is a path entirely of links that are postulated to be UP, so there is no cutset and

$$S_2 = (\overline{10}, 11, \overline{16}, 18, 20). \tag{B-16}$$

We examine W_3 given by (B-13). The forward search for a cutset finds that links 29 and 30 must be DOWN, while the reverse search finds that links 16 and 20 must be DOWN. The selected cutset is $(\overline{29}, \overline{30})$, giving rise to

$$F_6 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, \overline{30}) \tag{B-17}$$

and

$$Y_3 = (\overline{10}, 11, \overline{18}, 19, 28, 29) \tag{B-18a}$$

$$Y_4 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, 30). \tag{B-18b}$$

This exhausts the working set $\mathcal{W}$ given by (B-11) and (B-13). The temporary collection $\mathcal{Y}$ given by (B-15) and (B-18) has cardinality greater than zero. Therefore, we must make another pass through the algorithm with the new working collection given by (B-15) and (B-18) and $\mathcal{Y} \leftarrow \emptyset$.

We first examine W_1 given by (B-15a). The fact that links 11, 18, and 20 are postulated to be UP simultaneously guarantees a path, so

$$S_3 = (\overline{10}, 11, 16, 18, 20). \tag{B-19}$$

Next we examine W_2 given by (B-15b). Links 19, 21, and 22 must be DOWN to preclude a path, as determined by working forward from the source; the reverse cutset search finds that links 21 and 32 must be DOWN. Therefore, the selected cutset is ($\overline{21}$, $\overline{32}$), which gives rise to

$$F_7 = (\overline{10},\, 11,\, 16,\, 18,\, \overline{20},\, \overline{21},\, 24,\, \overline{32}) \tag{B-20}$$

and

$$Y_1 = (\overline{10},\, 11,\, 16,\, 18,\, \overline{20},\, 21,\, 24) \tag{B-21a}$$

$$Y_2 = (\overline{10},\, 11,\, 16,\, 18,\, \overline{20},\, \overline{21},\, 24,\, 32). \tag{B-21b}$$

We examine W_3 given by (B-18a). The candidates for cutset are ($\overline{20}$, $\overline{21}$, $\overline{30}$) from the forward search and ($\overline{16}$, $\overline{20}$) from the reverse search. Selecting the latter, we find the failure event

$$F_8 = (\overline{10},\, 11,\, \overline{16},\, \overline{18},\, 19,\, \overline{20},\, 28,\, 29) \tag{B-22}$$

and the temporary events

$$Y_3 = (\overline{10},\, 11,\, 16,\, \overline{18},\, 19,\, 28,\, 29) \tag{B-23a}$$

$$Y_4 = (\overline{10},\, 11,\, \overline{16},\, \overline{18},\, 19,\, 20,\, 28,\, 29). \tag{B-23b}$$

We examine W_4 given by (B-18b). The selected cutset (in the forward direction) is ($\overline{32}$), so to the failure collection we add

$$F_9 = (\overline{10},\, 11,\, \overline{18},\, 19,\, 28,\, \overline{29},\, 30,\, \overline{32}) \tag{B-24}$$

and to the temporary collection we add

$$Y_5 = (\overline{10},\, 11,\, \overline{18},\, 19,\, 28,\, \overline{29},\, 30,\, 32). \tag{B-25}$$

This exhausts the working set $\mathcal{W}$ given by (B-15) and (B-18). The temporary collection $\mathcal{Y}$ given by (B-21), (B-23), and (B-25) is not empty. Therefore we must make another pass through the algorithm with the new working collection $\mathcal{W}$ given by $\mathcal{W} \leftarrow \mathcal{Y}$, then set $\mathcal{Y} \leftarrow \emptyset$.

The set of good links postulated by W_1 given by (B-21a) forms a path from node 1 to node 2. Therefore, W_1 is a success event:

$$S_4 = (\overline{10},\, 11,\, 16,\, 18,\, \overline{20},\, 21,\, 24). \tag{B-26}$$

Next we examine W_2 given by (B-21b). The smallest cutset (found in the reverse direction) is ($\overline{30}$), giving rise to

$$F_{10} = (\overline{10},\, 11,\, 16,\, 18,\, \overline{20},\, \overline{21},\, 24,\, \overline{30},\, 32) \tag{B-27}$$

and

$$Y_1 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ \overline{21},\ 24,\ 30,\ 32). \tag{B-28}$$

The event W_3 given by (B-23a) cannot produce a path if either links 20, 21, and 30 are DOWN or links 20 and 24 are down. Selecting the latter (found by the reverse search) gives rise to

$$F_{11} = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{24},\ 28,\ 29) \tag{B-29}$$

and

$$Y_2 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ 20,\ 28,\ 29) \tag{B-30a}$$

$$Y_3 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ 24,\ 28,\ 29). \tag{B-30b}$$

Examining W_4 given by (B-23b), the conditions postulated guarantee a path:

$$S_5 = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 20,\ 28,\ 29). \tag{B-31}$$

Next we examine W_5 givien by (B-25). Either links 24 and 26 must be DOWN, as determined by a forward search, or links 16 and 20, as determined by a reverse search. Selecting the first alternative gives rise to

$$F_{12} = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{24},\ \overline{26},\ 28,\ \overline{29},\ 30,\ 32) \tag{B-32}$$

and

$$Y_4 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 24,\ 28,\ \overline{29},\ 30,\ 32) \tag{B-33a}$$

$$Y_5 = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{24},\ 26,\ 28,\ \overline{29},\ 30,\ 32). \tag{B-33b}$$

This exhausts the working set $\mathcal{W}$, with $\mathcal{Y}$ nonempty so that another iteration is required, using as the contents of $\mathcal{W}$ the events given by (B-28), (B-30), and (B-33). Considering W_1 given by (B-28), we find that in either search direction two links are required to be DOWN in order to preclude a path. Selecting the forward alternative, the cutset $(\overline{19},\ \overline{22})$ gives rise to

$$F_{13} = (\overline{10},\ 11,\ 16,\ 18,\ \overline{19},\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ 30,\ 32) \tag{B-34}$$

and

$$Y_1 = (\overline{10},\ 11,\ 16,\ 18,\ 19,\ \overline{20},\ \overline{21},\ 24,\ 30,\ 32) \tag{B-35a}$$

$$Y_2 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{19},\ \overline{20},\ \overline{21},\ 22,\ 24,\ 30,\ 32). \tag{B-35b}$$

The event W_2 given by (B-30a) supports a path of good links:

$$S_6 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ 20,\ 28,\ 29). \tag{B-36}$$

The event W_3 given by (B-30b), results in the cutset $(\overline{21},\ \overline{30})$ for the forward search. Thus

$$F_{14} = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ \overline{30}) \tag{B-37}$$

and

$$Y_3 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ 21,\ 24,\ 28,\ 29) \tag{B-38a}$$

$$Y_4 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ 30). \tag{B-38b}$$

The event W_4 given by (B-33a) results in the cutset ($\overline{16}$, $\overline{26}$) for a search in either direction. Thus

$$F_{15} = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 24,\ \overline{26},\ 28,\ \overline{29},\ 30,\ 32) \tag{B-39}$$

and

$$Y_5 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ 24,\ 28,\ \overline{29},\ 30,\ 32) \tag{B-40a}$$

$$Y_6 = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 24,\ 26,\ 28,\ \overline{29},\ 30,\ 32). \tag{B-40b}$$

Next we examine W_5 given by (B-33b). The cutset is ($\overline{20}$), leading to

$$F_{16} = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{20},\ \overline{24},\ 26,\ 28,\ \overline{29},\ 30,\ 32) \tag{B-41}$$

and

$$Y_7 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 20,\ \overline{24},\ 26,\ 28,\ \overline{29},\ 30,\ 32). \tag{B-42}$$

This exhausts the working set $\mathcal{W}$ and another iteration is required to examine the events given by (B-35), (B-38), (B-40) and (B-42). The event W_1 given by (B-35a) for a search in either direction results in the cutset ($\overline{22}$, $\overline{28}$), giving rise to

$$F_{17} = (\overline{10},\ 11,\ 16,\ 18,\ 19,\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ \overline{28},\ 30,\ 32) \tag{B-43}$$

and

$$Y_1 = (\overline{10},\ 11,\ 16,\ 18,\ 19,\ \overline{20},\ \overline{21},\ 22,\ 24,\ 30,\ 32) \tag{B-44a}$$

$$Y_2 = (\overline{10},\ 11,\ 16,\ 18,\ 19,\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ 28,\ 30,\ 32). \tag{B-44b}$$

The good links specified in events W_2 and W_3 given by (B-35b) and (B-38a), respectively, form complete paths:

$$S_7 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{19},\ \overline{20},\ \overline{21},\ 22,\ 24,\ 30,\ 32) \tag{B-45a}$$

$$S_8 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ 21,\ 24,\ 28,\ 29). \tag{B-45b}$$

Next we examine W_4 given by (B-38b) and find that the cutset is ($\overline{32}$), leading to

$$F_{18} = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ 30,\ \overline{32}) \tag{B-46}$$

and

$$Y_3 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ 30,\ 32). \tag{B-47}$$

There is no cutset for the event W_5 given by (B-40a), giving rise to

$$S_9 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ 24,\ 28,\ \overline{29},\ 30,\ 32). \tag{B-48}$$

The event W_6 given by (B-40b) has the cutset ($\overline{20}$), giving rise to

$$F_{19} = (\overline{10}, 11, \overline{16}, \overline{18}, 19, \overline{20}, 24, 26, 28, \overline{29}, 30, 32) \tag{B-49}$$

and

$$Y_4 = (\overline{10}, 11, \overline{16}, \overline{18}, 19, 20, 24, 26, 28, \overline{29}, 30, 32). \tag{B-50}$$

The good links in the event W_7 given by (B-42) form a path, so that

$$S_{10} = (\overline{10}, 11, \overline{18}, 19, 20, \overline{24}, 26, 28, \overline{29}, 30, 32). \tag{B-51}$$

A final iteration to examine the events in (B-44), (B-47), and (B-50) reveals that they are all success events:

$$S_{11} = (\overline{10}, 11, 16, 18, 19, \overline{20}, \overline{21}, 22, 24, 30, 32) \tag{B-52a}$$

$$S_{12} = (\overline{10}, 11, 16, 18, 19, \overline{20}, \overline{21}, \overline{22}, 24, 28, 30, 32) \tag{B-52b}$$

$$S_{13} = (\overline{10}, 11, 16, \overline{18}, 19, \overline{20}, \overline{21}, 24, 28, 29, 30, 32) \tag{B-52c}$$

$$S_{14} = (\overline{10}, 11, \overline{16}, \overline{18}, 19, 20, 24, 26, 28, \overline{29}, 30, 32). \tag{B-52d}$$

Both $\mathcal{W}$ and the temporary collection $\mathcal{Y}$ are empty, so we are done! The collection $\mathcal{S}$ is now an exhaustive success collection and the collection $\mathcal{F}$ is an exhaustive failure collection. The probability of successfully routing a call from source 1 to destination 2 (s-t reliability with s $= 1$ and t $= 2$) may be computed using either $\mathcal{S}$ or $\mathcal{F}$.

APPENDIX C

EXAMPLE OF A PROBABILITY CALCULATION FOR A 3 × 3 NETWORK USING THE CUTPATH METHOD (ORIGIN = 1, DESTINATION = 2)

In this appendix an example is provided of the operation of the method for the s-t reliability calculation in which a selection is made of either a path or cutset, depending upon which will generate the fewer number of new events. The problem considered is that of finding the probability that, in the 3 × 3 network of Figure A-1, the flood search succeeds in finding a path from node 1 to node 2. This is the same example worked for the equivalent-links algorithm with pathfinding in Appendix A and using cutsets in Appendix B.

The processing begins by setting $\mathcal{W} = \{W_1 = [0\ \ 0\ \ 0 \cdots 0]\}$, the universal event. In accordance with the flow diagram of Figure 2-9, first a short path from node 1 to node 2 is sought. Using an unspecified pathfinding algorithm and an unspecified minimum cutset algorithm that works both forward from the source and backward from the sink—and giving preference to a cutset found in a forward search in case of a tie in the size of the cutsets found—the problem proceeds as follows, assuming that in the case of a tie in the number of new events to be generated, a path is preferred.

The source is $s = 1$ and the terminal is $t = 2$. In the forward direction the smallest path is (10) and the smallest cutset is ($\overline{10}$, $\overline{11}$); that is, there is no path out from the source if links 10 and 11 are DOWN. In the reverse direction the smallest cutset is ($\overline{10}$, $\overline{16}$, $\overline{20}$); that is, there is no path into the sink if links 10, 16, and 20 are DOWN. Thus the selection is pathfinding, with

$$P = (10) \tag{C-1}$$

which includes, in this instance, the link which directly connects s and t. Since we have successfully found a path, we add $W_1 \cap P$ to $\mathcal{S}$:

$$S_1 = (10) \tag{C-2}$$

and add the complement, $W_1 \cap \overline{S}$ to $\mathcal{Y}$[12]:

$$Y_1 = (\overline{10}). \tag{C-3}$$

[12]Only one first-in-first-out queue is required. For ease of presentation we speak of the collection $\mathcal{W}$ of events being read from the queue, and the collection $\mathcal{Y}$ of events being added to the queue.

The initial working set $\mathcal{W}$ is now exhausted. Since $|\mathcal{Y}| = 1$, the problem is not done. Therefore the queues are exchanged by setting $\mathcal{W} \leftarrow \mathcal{Y}$, *i. e.*, by setting

$$W_1 = (\overline{10}) \tag{C-4}$$

and then setting

$$\mathcal{Y} = \emptyset. \tag{C-5}$$

On the second cycle through the algorithm, $|\mathcal{W}| = 1$, so one there is one event to be examined. Examining W_1, it is found that the shortest path requires three links: $P = (11, 18, 20)$, which would eventuate in the generation of three new events to be examined. In search of a short cutset, it is found that, given that link 10 is DOWN, only link 11 has to be down to preclude a path from $s = 1$ to $t = 2$. Thus the cutset method is selected, with

$$C = (\overline{11}), \tag{C-6}$$

giving rise to the failure event $W_1 \cap C$, or

$$F_1 = (\overline{10}, \overline{11}), \tag{C-7}$$

which is appended to $\mathcal{F}$, and the next event $W_1 \cap \overline{C}$, or

$$Y_1 = (\overline{10}, 11). \tag{C-8}$$

The working set $\mathcal{W}$ is now exhausted. Since $|\mathcal{Y}| = 1 > 0$, another cycle through the algorithm is needed. The new working set is obtained by setting $\mathcal{W}$ equal to the temporary collection $\mathcal{Y}$ as given in (C-8):

$$W_1 = (\overline{10}, 11). \tag{C-9}$$

The next cycle through the algorithm begins by examining W_1. Again, $P = (11, 18, 20)$ is the shortest path; this time the pathfinding method would generate two new events, since only links 18 and 20 are not specified to be UP in W_1. In the forward direction, links 18 and 19 must be DOWN to preclude a path from node 1 to node 2, and links 16 and 20 in the reverse direction. Therefore, the selected method is to use pathfinding, giving rise to

$$S_2 = (\overline{10}, 11, 18, 20) \tag{C-10}$$

and

$$Y_1 = (\overline{10}, 11, \overline{18}) \tag{C-11a}$$

$$Y_2 = (\overline{10}, 11, 18, \overline{20}). \tag{C-11b}$$

On the next pass through the algorithm, W_1 given by (C-11a) is examined. With link 18 DOWN, the shortest path is $P = (11, 19, 28, 29, 20)$, involving four links not

specified to be UP in W_1. The forward cutset involves links only one link, link 19, and therefore is the preferred cutset, giving rise to

$$F_2 = (\overline{10},\ 11,\ \overline{18},\ \overline{19}) \tag{C-12}$$

and

$$Y_1 = (\overline{10},\ 11,\ \overline{18},\ 19). \tag{C-13}$$

Examination of W_2 given by (C-11b) determines that the least number of new events is generated using the cutset $C = (\overline{16})$, giving rise to

$$F_3 = (\overline{10},\ 11,\ \overline{16},\ 18,\ \overline{20}) \tag{C-14}$$

and

$$Y_2 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20}). \tag{C-15}$$

This exhausts the working collection (C-11). Since $|\mathcal{Y}| > 0$, another pass through the algorithm is needed with the new working set given by the events in (C-13) and (C-15).

On examination of W_1 given by (C-13), it is found that in the forward direction, only link 28 must be DOWN to preclude a path. Therefore the selected cutset is $(\overline{28})$, giving rise to

$$F_4 = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{28}) \tag{C-16}$$

and

$$Y_1 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 28). \tag{C-17}$$

W_2 given by (C-15). Similarly, there is no path if only one link is down, link 24. Thus

$$F_5 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ \overline{24}) \tag{C-18}$$

and

$$Y_2 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ 24). \tag{C-19}$$

This exhausts the working set $\mathcal{W}$ given by (C-13) and (C-15). The temporary collection $\mathcal{Y}$ given by (C-17) and (C-19) has cardinality greater than zero. Therefore, another pass through the algorithm is required, with the new working collection given by (C-17) and (C-19) and $\mathcal{Y} \leftarrow \emptyset$.

Examining W_1 given by (C-17) determines that the shortest path $P = (11,\ 19,\ 28,\ 29,\ 20)$ would generate two new events, as would the smallest cutset, $C = (\overline{16},\ \overline{20})$. Therefore the choice is pathfinding, giving

$$S_3 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 20,\ 28,\ 29) \tag{C-20}$$

with

$$Y_1 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}) \tag{C-21a}$$

$$Y_2 = (\overline{10}, 11, \overline{18}, 19, \overline{20}, 28, 29). \tag{C-21b}$$

Next W_2 given by (C-19) is examined. The path $P = (11, 18, 21, 24, 16)$ only generates one new event, so it is not necessary to find a cutset. The ELA success event processing gives

$$S_4 = (\overline{10}, 11, 16, 18, \overline{20}, 21, 24) \tag{C-22}$$

and

$$Y_3 = (\overline{10}, 11, 16, 18, \overline{20}, \overline{21}, 24). \tag{C-23}$$

This exhausts the working set $\mathcal{W}$ given by (C-17) and (C-19). The temporary collection $\mathcal{Y}$ given by (C-21) and (C-23) is not empty. Therefore another pass through the algorithm is required.

The shortest path under the network condition postulated by W_1 given by (C-21a) would add four new events, while $C = (\overline{30})$ is a cutset. Therefore, W_1 is an ELA2 failure event:

$$F_6 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, \overline{30}), \tag{C-24}$$

giving rise to

$$Y_1 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, 30). \tag{C-25}$$

The shortest path for W_2 given by (C-21b) would use three links not said to UP in W_2. The smallest cutset (found in the reverse direction) is $(\overline{16})$, giving rise to

$$F_7 = (\overline{10}, 11, \overline{16}, \overline{18}, 19, \overline{20}, 28, 29) \tag{C-26}$$

and

$$Y_2 = (\overline{10}, 11, 16, \overline{18}, 19, \overline{20}, 28, 29). \tag{C-27}$$

The event W_3 given by (C-23) cannot produce a path if either links 19 and 22 are DOWN or link 32 is down. Selecting the latter (found by the reverse search) gives

$$F_8 = (\overline{10}, 11, 16, 18, \overline{20}, \overline{21}, 24, \overline{32}) \tag{C-28}$$

and

$$Y_3 = (\overline{10}, 11, 16, 18, \overline{20}, \overline{21}, 24, 32). \tag{C-29}$$

This exhausts the working set $\mathcal{W}$, with $\mathcal{Y}$ nonempty so that another iteration is required, using as the contents of $\mathcal{W}$ the events given by (C-25), (C-27), and (C-29). Considering W_1 given by (C-25), the shortest path adds three new links, while the cutset $C = (\overline{32})$ is found, giving rise to

$$F_9 = (\overline{10}, 11, \overline{18}, 19, 28, \overline{29}, 30, \overline{32}) \tag{C-30}$$

and

$$Y_1 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 28,\ \overline{29},\ 30,\ 32). \tag{C-31}$$

The event W_2 given by (C-27) will not support a path (the shortest of which involves two new links) if link 24 is DOWN, so there are developed

$$F_{10} = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{24},\ 28,\ 29) \tag{C-32}$$

and

$$Y_2 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ 24,\ 28,\ 29). \tag{C-33}$$

The event W_3 given by (C-29) results in selection of the cutset ($\overline{30}$) from the reverse search. Thus

$$F_{11} = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ \overline{21},\ 24,\ \overline{30},\ 32) \tag{C-34}$$

and

$$Y_3 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ \overline{21},\ 24,\ 30,\ 32). \tag{C-35}$$

Once again the working set $\mathcal{W}$ is exhausted and another iteration is required to examine the events given by (C-31), (C-33), and (C-35). The event W_1 given by (C-31) for a cutset search in either direction would generate two new events, the same number of new events for the path $P = (11,\ 19,\ 28,\ 30,\ 32,\ 24,\ 16)$. Thus pathfinding is preferred, giving

$$S_5 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ 24,\ 28,\ \overline{29},\ 30,\ 32) \tag{C-36}$$

and

$$Y_1 = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{24},\ 28,\ \overline{29},\ 30,\ 32) \tag{C-37a}$$

$$Y_2 = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 24,\ 28,\ \overline{29},\ 30,\ 32). \tag{C-37b}$$

Next examined is W_2 given by (C-33), for which the shortest path is found to add only new event, involving link 21. It therefore is not necessary to search for a cutset. The success event and the new event generated are

$$S_6 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ 21,\ 24,\ 28,\ 29) \tag{C-38}$$

and

$$Y_3 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29). \tag{C-39}$$

Similarly, only link 22 is new for the path found for W_3 given by (C-35), giving

$$S_7 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ \overline{21},\ 22,\ 24,\ 30,\ 32) \tag{C-40}$$

and

$$Y_4 = (\overline{10},\ 11,\ 16,\ 18,\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ 30,\ 32). \tag{C-41}$$

Another iteration is needed to examine the events in (C-37), (C-39), and (C-41). In each case, a cutset of only one link failure is found giving rise to the ELA2 failure events

$$F_{12} = (\overline{10},\ 11,\ \overline{18},\ 19,\ \overline{20},\ \overline{24},\ 28,\ \overline{29},\ 30,\ 32) \tag{C-42a}$$

$$F_{13} = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ \overline{20},\ 24,\ 28,\ \overline{29},\ 30,\ 32) \tag{C-42b}$$

$$F_{14} = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ \overline{30}) \tag{C-42c}$$

$$F_{15} = (\overline{10},\ 11,\ 16,\ 18,\ \overline{19},\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ 30,\ 32) \tag{C-42d}$$

and the next events

$$Y_1 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 20,\ \overline{24},\ 28,\ \overline{29},\ 30,\ 32) \tag{C-43a}$$

$$Y_2 = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 20,\ 24,\ 28,\ \overline{29},\ 30,\ 32) \tag{C-43b}$$

$$Y_3 = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ 30) \tag{C-43c}$$

$$Y_4 = (\overline{10},\ 11,\ 16,\ 18,\ 19,\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ 30,\ 32). \tag{C-43d}$$

The next pass of the algorithm uses (C-43) as $\mathcal{W}$. All the events examined lead to paths with just one new link, so the ELA success event processing is used, giving rise to

$$S_8 = (\overline{10},\ 11,\ \overline{18},\ 19,\ 20,\ \overline{24},\ 26,\ 28,\ \overline{29},\ 30,\ 32) \tag{C-44a}$$

$$S_9 = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 20,\ 24,\ 26,\ 28,\ \overline{29},\ 30,\ 32) \tag{C-44b}$$

$$S_{10} = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ 30,\ 32) \tag{C-44c}$$

$$S_{11} = (\overline{10},\ 11,\ 16,\ 18,\ 19,\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ 28,\ 30,\ 32) \tag{C-44d}$$

and four new events which on the final pass of the algorithm, determined to be ELA failure events (no paths):

$$F_{16} = (\overline{10},\ 11,\ \overline{18},\ 19,\ 20,\ \overline{24},\ \overline{26},\ 28,\ \overline{29},\ 30,\ 32) \tag{C-45a}$$

$$F_{17} = (\overline{10},\ 11,\ \overline{16},\ \overline{18},\ 19,\ 20,\ 24,\ \overline{26},\ 28,\ \overline{29},\ 30,\ 32) \tag{C-45b}$$

$$F_{18} = (\overline{10},\ 11,\ 16,\ \overline{18},\ 19,\ \overline{20},\ \overline{21},\ 24,\ 28,\ 29,\ 30,\ \overline{32}) \tag{C-45c}$$

$$F_{19} = (\overline{10},\ 11,\ 16,\ 18,\ 19,\ \overline{20},\ \overline{21},\ \overline{22},\ 24,\ \overline{28},\ 30,\ 32). \tag{C-45d}$$

The collection $\mathcal{S}$ is now an exhaustive success collection and the collection $\mathcal{F}$ is an exhaustive failure collection. The probability of successfully routing a call from source 1 to destination 2 (s-t reliability with $s = 1$ and $t = 2$) may be computed using either $\mathcal{S}$ or $\mathcal{F}$.

APPENDIX D
COMPUTER PROGRAM LISTINGS

In this appendix the major programs referred to in the text are listed in order to document the project. All programs are in Turbo-Pascal and were developed in the Turbo-Pascal version 6.0 programming environment.

D.1 IMPLEMENTATION OF THE ELA

D.1.1 **Program EQLNKTST.PAS**

```
PROGRAM EQLNKTST;
USES
    Crt, Dos, Eqlinks, PopMenus, Dir_Menu, Strings, Keyboard, FileChck,
    Netset;
VAR
   NW : Network;
   NumNodes, NumJams, kj, hopmax : integer;
   Thr, Sig : real;
(* ------------------------------------------------------------------- *)
PROCEDURE InitializeSINR(VAR SINR : ThreeDarray);
CONST
  BIGNEG = -99.9;
VAR
  i, j, k : Integer;
BEGIN
  FOR i := 1 TO maxv DO
    FOR j := 1 TO maxv DO
      FOR k := 1 TO maxv DO
        SINR[i,j,k] := BIGNEG;
END; (* InitializeSINR *)
(* ------------------------------------------------------------------- *)
PROCEDURE GetSNRs(VAR SINR : ThreeDarray);
LABEL
  READ_ERROR;
VAR
  Msg, s, FileSpec, FileStr, ExtStr, SNRStr, DirSpec : String;
  SNRFile : Text;
  i, j, k : integer;
BEGIN
  InitializeSINR(SINR);
  FileSpec := '*.SNR';
  DirSpec := '';
  Msg := '  << SNRFile Selection (F1 for HELP)';
  SNRStr := DirectoryMenu(DirSpec, FileSpec, Msg);
  MakeStrUpper(SNRStr);
  {$V-} Fsplit(SNRStr, DirSpec, FileStr, ExtStr); {$V+}
  IF NOT CheckOldFile(SNRFile, SNRStr) THEN
    GoTo READ_ERROR;
  {$I-} Readln(SNRFile, s); {$I+}
  IF (IOResult <> 0) OR (ExitCode <> 0) THEN GoTO READ_ERROR;
  {$I-} Readln(SNRFile, i, NumNodes, NumJams); {$I+}
  WHILE NOT EOF(SNRFile) DO
    BEGIN
      {$I-} Read(SNRFile, i, j); {$I+}
      IF NumJams = 1 THEN
```

```
        {$I-} Readln(SNRFile, SINR[i,j,1]) {$I+}
      ELSE
        BEGIN
          FOR k := 1 to NumJams - 1 DO
            {$I-} Read(SNRFile, SINR[i,j,k]); {$I+}
          {$I-} Readln(SNRFile, SINR[i,j,NumJams]); {$I+}
        END;
    END; (* WHILE *)
  Close(SNRFile);
  Exit;
READ_ERROR:
  Close(SNRFile);
END; (* GetSNRs *)
(* ------------------------------------------------------------------ *)
PROCEDURE GetData;
BEGIN
    ClrScr;
    QuickPopUp(20, 5, 60, 15, 2, White, Blue, '');
    Writeln;
    Write(' Enter threshold value in dB (0): ');
    Readln(Threshold);
    Writeln;
    Write(' Enter propagation sigma in dB (10) : ');
    Readln(SigmaL);
    ClosePopUp;
END; (* GetData *)
(* ------------------------------------------------------------------ *)
PROCEDURE Testbeta(VAR SINR : ThreeDarray);
VAR
  s, More : String;
  i, j, k : integer;
BEGIN
  More := 'Y';
  WHILE More = 'Y' DO
  BEGIN
    ClrScr;
    QuickPopUp(20, 5, 60, 15, 2, Black, Cyan, '');
    Writeln;
    Write('  Calculate a link reliability (Y/N) ? ');
    Readln(s);
    MakeStrUpper(s);
    More := s;
    IF More = 'Y' THEN
      BEGIN
        Writeln;
        Write(' Enter index of source node (1-',NumNodes,'): ');
        Readln(i);
        Writeln;
        Write(' Enter index of destination node : ');
        Readln(j);
        Writeln;
        Write(' Enter jammer case (1-',NumJams,') : ');
        Readln(k);
        Writeln;
        Writeln(' Beta equals ',
                    Pfunction((SINR[i,j,k]-Threshold)/SigmaL));
        Writeln;
        Pause('  Press any key to continue...');
      END;
    ClosePopUp;
```

```
  END; (* WHILE *)
END; (* Testbeta *)
(* ------------------------------------------------------------------ *)
PROCEDURE TestSTRel(VAR NW : Network);
VAR
  s, More : String;
  PL, PU : Real;
BEGIN
  More := 'Y';
  WHILE More = 'Y' DO
  BEGIN
    ClrScr;
    QuickPopUp(20, 5, 60, 18, 2, Black, Yellow, '');
    Writeln;
    Write('  Calculate an ST Reliability (Y/N) ? ');
    Readln(s);
    MakeStrUpper(s);
    More := s;
    IF More = 'Y' THEN
      BEGIN
        Writeln;
        Write(' Enter index of source node (1-',NumNodes,'): ');
        Readln(orig);
        Writeln;
        Write(' Enter index of destination node : ');
        Readln(dest);
        Writeln;
        Write(' Enter maximum no. of hops (10) : ');
        Readln(hopmax);
        Writeln;
        Write(' Enter jammer case (1-',NumJams,') : ');
        Readln(kj);
        GetNett(NW, NumNodes, kj, hopmax, Threshold, SigmaL);
        ELReliabil(NW, PL, PU);
        Writeln;
        Writeln(' Lower bound: ', PL);
        Writeln;
        Writeln(' Upper bound: ', PU);
        Writeln;
        Pause('  Press a key to continue...');
      END;
    ClosePopUp;
  END; (* WHILE *)
END; (* TestSTRel *)
(* ---------------------- Main Program ----------------------------- *)
BEGIN
  ClrScr;
  GetSNRs(SINR);
  GetData;
  Testbeta(SINR);
  OpenNetwork(NW);
  TestSTRel(NW);
  CloseNetwork(NW);
END. (* Main Program *)
```

D.1.2 **Unit EQLINKS.PAS**

```
UNIT EQLINKS;
```

```
INTERFACE
USES
    Netset;
CONST
    maxv = 15;          {Maximum number of vertices in the graph}[13]
    maxe = 225;          {Maximum number of edges}
    maxjam = 10;           {Maximum number of jammers}
TYPE
    Vectj = array[1..maxjam] of real;
    Matrxi = array[1..maxv] of Vectj;
    ThreeDarray = array[1..maxv] of Matrxi;
VAR
    SINR, BetaAll : ThreeDarray;
    Threshold, SigmaL : Real;
    Orig, Dest : integer;
FUNCTION Pfunction(z : real) : real;
PROCEDURE SwapFiles(VAR OldQ, NewQ : Text);
PROCEDURE SetParam(NW : Network);
PROCEDURE Success(VAR NW : Network; s : String; VAR NewCount : integer);
PROCEDURE Failure(VAR NW : Network; s : String);
PROCEDURE ProcessQ(VAR NW : Network);
PROCEDURE Cleanup(VAR NW : Network);
PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
            threshold, sigmaL : real);
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU : real);
(* ------------------------------------------------------------------------
*)
IMPLEMENTATION
USES
    Crt, DOS, FileChck;
CONST
   EVENTMIN = 98;
   EVENTMAX = 39998;
   EPSILON = 1.0E-2;
VAR
    NW : Network;              {Network graph}
    origin, destination, hopmax : integer;
    NumNodes : integer;  {maximum node index used}
    OldQ, NewQ : Text;
    ReadStr, WriteStr : string;
    QueueSize, TotCount, SuccCount, FailCount : LongInt;
    SuccProb, FailProb : real;
(* ------------------------------------------------------------------------
*)
FUNCTION Pfunction(z : real) : real;
VAR
  t, temp : real;
BEGIN
  t := 1.0/(1.0 + 0.33267*abs(z));
  temp := 0.4361836*t - 0.1201676*sqr(t) + 0.937298*t*sqr(t);
  IF abs(z) > 20.0 THEN
    temp := 0.0
  ELSE
    temp := temp * exp(-0.5*sqr(z))/Sqrt(2.0*pi);
  IF z >= 0.0 THEN
    Pfunction := 1.0 - temp
  ELSE
```

[13]These limits can be changed as appropriate when running the program from TurboPascal.

```
    Pfunction := temp;
END; (* Pfunction *)
(* ------------------------------------------------------------------------
*)
PROCEDURE SetParam(NW : Network);
VAR
  Event : string;
  Happen : boolean;
BEGIN
  TotCount := 0;
  SuccCount := 0;
  FailCount := 0;
  QueueSize := 1;
  SuccProb := 0.0;
  FailProb := 0.0;
  ReadStr := 'DATA2.TMP';
  WriteStr := 'DATA1.TMP';
  happen := CheckNewFile(NewQ, WriteStr);
  happen := CheckNewFile(OldQ, ReadStr);
  WITH NW DO
    BEGIN
      InitialEvent(Event, '1', EdgeNum);
      SaveEvent(NewQ, NW, Event);
      SwapFiles(OldQ, NewQ);
      SaveEvent(NewQ, NW, Event);
    END; (* WITH *)
END; (* SetParam *)
(* ------------------------------------------------------------------------
*)
PROCEDURE SwapFiles(VAR OldQ, NewQ : Text);
VAR
 s : string;
 happen : boolean;
BEGIN
  Close(NewQ);
  Close(OldQ);
  s := ReadStr;
  ReadStr := WriteStr;
  WriteStr := s;
  happen := CheckOldFile(OldQ, ReadStr);
  happen := CheckNewFile(NewQ, WriteStr);
END; (* SwapFiles *)
(* ------------------------------------------------------------------------
*)
PROCEDURE Success(VAR NW : Network; s : string; VAR NewCount : integer);
VAR
  SuccEvent : string;
  i : integer;
BEGIN
 WITH NW DO
   BEGIN
     NewCount := 0;
     SuccEvent[0] := Char(EdgeNum);
     FOR i := 1 TO EdgeNum DO
       BEGIN
         IF UpEdges[i] = '1' THEN
           SuccEvent[i] := s[i]
         ELSE
           SuccEvent[i] := UpEdges[i];
```

```
          IF (s[i] = '1') AND (UpEdges[i] <> '1') THEN
            Inc(NewCount);
        END; (* FOR *)
   END; (* WITH *)
 SuccProb := SuccProb + ProbEvent(NW, SuccEvent);
 Inc(SuccCount);
 Inc(TotCount);
END; (* Success *)
(* ------------------------------------------------------------------
*)
PROCEDURE Failure(VAR NW : Network; s: string);
BEGIN
  FailProb := FailProb + ProbEvent(NW, s);
  Inc(FailCount);
  Inc(TotCount);
END; (* Failure *)
(* ------------------------------------------------------------------
*)
PROCEDURE ProcessQ(VAR NW : Network);
VAR
  Event : String;
  PathLen, NewCount : Integer;
BEGIN
  QueueSize := 0;
  SwapFiles(OldQ, NewQ);
  WHILE NOT EOF(OldQ) DO
    BEGIN
      ReadEvent(OldQ, Event);
      SetEventLinks(NW, Event);
      IF TwoWaySearch(NW, PathLen) THEN
        BEGIN (* Path found *)
          Success(NW, Event, NewCount);
          ComplementEvent(NewQ, NW, Event, NewCount);
          Inc(QueueSize, NewCount);
        END (* Path found *)
      ELSE
        Failure(NW, Event);
      RestoreEventLinks(NW);
      IF TotCount > EVENTMAX THEN Exit;
      IF ((1.0 - (SuccProb + FailProb)) < EPSILON) AND
         (TotCount > EVENTMIN) THEN Exit;
    END; (* WHILE *)
END; (* ProcessQ *)
(* ------------------------------------------------------------------
*)
PROCEDURE Cleanup(VAR NW : Network);
BEGIN
  Close(NewQ);
  Close(OldQ);
  Erase(NewQ);
  Erase(OldQ);
END; (* Cleanup *)
(* ------------------------------------------------------------------
*)
PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
                  Threshold, SigmaL : real);
   {Initialize the network NW from SNRs supplied in array SINR.}
VAR
   k, n, nv, sv, tv : integer;
   arg : real;
```

```
BEGIN  {GetNett}
  WITH NW DO
  BEGIN
    n := 0;
    nv := NumNodes;
    MaxHops := hopmax;
    NodeNum := NumNodes;
    FOR sv := 1 TO nv DO BEGIN  {loop on source vertex}
    FOR tv := 1 TO nv DO BEGIN  {loop on terminal vertex}
      IF sv <> tv THEN
        BEGIN  {create an edge}
          arg := (SINR[sv, tv, kj] - Threshold)/SigmaL;
          BetaAll[sv, tv, kj] := Pfunction(arg);
          IF arg >= 0.0 THEN
          BEGIN  {there is a viable edge}
            n := n + 1;
            I_Index^[n] := sv;
            J_Index^[n] := tv;
            Beta^[n] := BetaAll[sv, tv, kj];
            GraphMat^[sv,tv] := n;
          END {Viable edges}
          ELSE GraphMat^[sv,tv] := 0;
        END;  {create an edge}
    END;    {loop on terminal vertex}
    END;      {loop on source vertex}
    Edgenum := n;
  END; (* WITH *)
END;  {GetNett}
(* ------------------------------------------------------------------------
*)
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU : real);
BEGIN
  WITH NW DO
  BEGIN
    source := orig;
    sink := dest;
    SetParam(NW);
    WHILE QueueSize <> 0 DO
      ProcessQ(NW);
    PL := SuccProb;
    PU := 1.0 - FailProb;
    Cleanup(NW);
  END; (* WITH *)
END; (* STReliabil *)
END.
```

D.2 IMPLEMENTATION OF THE ELA 2

D.2.1 **Program EL2ONLY.PAS**

```
PROGRAM EL2only;
USES
    Crt, Dos, Cutonly, PopMenus, Dir_Menu, Strings, Keyboard, FileChck,
    NetSet;
VAR
   NW : Network;
   NumNodes, NumJams, kj, hopmax : integer;
```

```
   Thr, Sig : real;
(* ---------------------------------------------------------------------- *)
PROCEDURE InitializeSINR(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ---------------------------------------------------------------------- *)
PROCEDURE GetSNRs(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ---------------------------------------------------------------------- *)
PROCEDURE GetData;
```

[For a listing of this procedure, see Section D.1.1]

```
(* ---------------------------------------------------------------------- *)
PROCEDURE Testbeta(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ---------------------------------------------------------------------- *)
PROCEDURE TestSTRel(VAR NW : Network);
VAR
  s, More : String;
  PL, PU, PE : Real;
BEGIN
  More := 'Y';
  WHILE More = 'Y' DO
  BEGIN
    ClrScr;
    QuickPopUp(20, 5, 60, 18, 2, Black, Yellow, '');
    Writeln;
    Write('  Calculate an ST Reliability (Y/N) ? ');
    Readln(s);
    MakeStrUpper(s);
    More := s;
    IF More = 'Y' THEN
      BEGIN
        Writeln;
        Write(' Enter index of source node (1-',NumNodes,'): ');
        Readln(orig);
        Writeln;
        Write(' Enter index of destination node : ');
        Readln(dest);
        Writeln;
        Write(' Enter maximum no. of hops (10) : ');
        Readln(hopmax);
        Writeln;
        Write(' Enter jammer case (1-',NumJams,') : ');
        Readln(kj);
        GetNett(NW, NumNodes, kj, hopmax, Threshold, SigmaL);
        ELReliabil(NW, PL, PU, PE);
        Writeln;
        Writeln(' Lower bound: ', PL);
        Writeln;
        Writeln(' Upper bound: ', PU);
        Writeln;
        Writeln(' Estimate:    ', PE);
        Writeln;
        Pause('  Press a key to continue...');
      END;
```

```
    ClosePopUp;
  END; (* WHILE *)
END; (* TestSTRel *)
(* ---------------------- Main Program ------------------------------ *)
BEGIN
  ClrScr;
  GetSNRs(SINR);
  GetData;
  Testbeta(SINR);
  OpenNetwork(NW);
  TestSTRel(NW);
  CloseNetwork(NW);
END. (* Main Program *)
```

D.2.2 UNIT CUTONLY.PAS

```
UNIT CUTonly;
INTERFACE
USES
    NetSet;
CONST
    maxv = 20;          {Maximum number of vertices in the graph}
    maxe = 255;          {Maximum number of edges}
    maxjam = 10;           {Maximum number of jammers}
TYPE
    Vectj = array[1..maxjam] of real;
    Matrxi = array[1..maxv] of Vectj;
    ThreeDarray = array[1..maxv] of Matrxi;
VAR
    SINR, BetaAll : ThreeDarray;
    Threshold, SigmaL : Real;
    Orig, Dest : integer;
    Fname : string;
    Fileout : text;
FUNCTION Pfunction(z : real) : real;
PROCEDURE SwapFiles(VAR OldQ, NewQ : Text);
PROCEDURE SetParam(NW : Network);
PROCEDURE NoCut(VAR NW : Network; s: String);
PROCEDURE CutFail(VAR NW : Network; s: String; VAR NewCount : integer);
PROCEDURE ProcessQ(VAR NW : Network);
PROCEDURE Cleanup(VAR NW : Network);
PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
            threshold, sigmaL : real);
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU, PE : real);
(* ------------------------------------------------------------------
*)
IMPLEMENTATION
USES
    Crt, DOS, FileChck;
CONST
   EVENTMIN = 98;
   EVENTMAX = 39998;
   EPSILON = 1.0E-2;
VAR
    NW : Network;              {Network graph}
    origin, destination, hopmax : integer;
    NumNodes : integer;  {maximum node index used}
    OldQ, NewQ : Text;
```

```
    ReadStr, WriteStr : string;
    QueueSize, TotCount, SuccCount, FailCount : LongInt;
    SuccProb, FailProb, Estimate : real;
(* ------------------------------------------------------------------------
*)
FUNCTION Pfunction(z : real) : real;
```

[For a listing of this function, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE SetParam(NW : Network);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE SwapFiles(VAR OldQ, NewQ : Text);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE NoCut(VAR NW : Network; s : String);
BEGIN
  SuccProb := SuccProb + ProbEvent(NW, s);
  Inc(SuccCount);
  Inc(TotCount);
END; (* NoCut *)
(* ------------------------------------------------------------------------
*)
PROCEDURE CutFail(VAR NW : Network; s : String; VAR NewCount : Integer);
VAR
  CutEvent : String;
  i : Integer;
BEGIN
  WITH NW DO
  BEGIN
    NewCount := 0;
    CutEvent[0] := Char(EdgeNum);
    FOR i := 1 TO EdgeNum DO
      BEGIN
        IF UpEdges[i] = '1' THEN CutEvent[i] := s[i]
        ELSE CutEvent[i] := '0';
        IF UpEdges[i]<>'1' THEN Inc(NewCount);
      END; (* FOR *)
  END; (* WITH *)
  FailProb := FailProb + ProbEvent(NW, CutEvent);
  Inc(FailCount);
  Inc(TotCount);
END; (* CutFail *)
(* ------------------------------------------------------------------------
*)
PROCEDURE ProcessQ(VAR NW : Network);
VAR
  Event : String;
  PathLen, NewCount : Integer;
BEGIN
  QueueSize := 0;
  SwapFiles(OldQ, NewQ);
  WHILE NOT EOF(OldQ) DO
    BEGIN
```

```
      ReadEvent(OldQ, Event);
      SetEventLinks(NW, Event);
      IF CutSetSearch(NW, Event) THEN
        BEGIN (* Cut Set found *)
          CutFail(NW, Event, NewCount);
          AddEvents(NewQ, NW, Event, NewCount);
          Inc(QueueSize, NewCount);
         { Estimate := 0.5*(1.0 + SuccProb - FailProb);
          Writeln(fileout, totcount:5,succprob:9:6,
                (1.0-failprob):9:6, estimate:9:6);}
        END (* Cut Set found *)
      ELSE
        NoCut(NW, Event);
      RestoreEventLinks(NW);
      IF TotCount > EVENTMAX THEN Exit;
      IF ((1.0 - (SuccProb + FailProb)) < EPSILON) AND
         (succcount + failcount > EVENTMIN) THEN Exit;
    END; (* WHILE *)
END; (* ProcessQ *)
(* ------------------------------------------------------------------------
*)
PROCEDURE Cleanup(VAR NW : Network);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
                  Threshold, SigmaL : real);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU, PE : real);
BEGIN
  WITH NW DO
  BEGIN
    source := orig;
    sink := dest;
    SetParam(NW);
    WHILE (QueueSize <> 0) DO
      ProcessQ(NW);
    PL := SuccProb;
    PU := 1.0 - FailProb;
    PE := 0.5*(PL + PU);
    Cleanup(NW);
  END; (* WITH *)
END; (* ELReliabil *)
END.
```

D.3 IMPLEMENTATION OF COMBINED ELA AND ELA2

D.3.1 **Program EL1&2.PAS**

```
PROGRAM EL1and2;
USES
```

```
    Crt, Dos, CutSet, PopMenus, Dir_Menu, Strings, Keyboard, FileChck,
    NetSet;
VAR
   NW : Network;
   NumNodes, NumJams, kj, hopmax : integer;
   Thr, Sig : real;
(* ------------------------------------------------------------------ *)
PROCEDURE InitializeSINR(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE GetSNRs(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE GetData;
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE Testbeta(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE TestSTRel(VAR NW : Network);
```

[For a listing of this procedure, see Section D.2.1]

```
(* ---------------------- Main Program ----------------------------- *)
BEGIN
  ClrScr;
  GetSNRs(SINR);
  GetData;
  Testbeta(SINR);
  OpenNetwork(NW);
  TestSTRel(NW);
  CloseNetwork(NW);
END. (* Main Program *)
```

D.3.2 Unit CUTSET.PAS

```
UNIT CUTSET;
INTERFACE
USES
    NetSet;
CONST
    maxv = 35;          {Maximum number of vertices in the graph}
    maxe = 127;          {Maximum number of edges}
    maxjam = 4;           {Maximum number of jammers}
TYPE
    Vectj = array[1..maxjam] of real;
    Matrxi = array[1..maxv] of Vectj;
    ThreeDarray = array[1..maxv] of Matrxi;
VAR
    SINR, BetaAll : ThreeDarray;
    Threshold, SigmaL : Real;
    Orig, Dest : integer;
    Fname : string;
```

```
    Fileout : text;
FUNCTION Pfunction(z : real) : real;
PROCEDURE SwapFiles(VAR OldQ, NewQ, OldQC, NewQC : Text);
PROCEDURE SetParam(NW : Network);
PROCEDURE Success(VAR NW : Network; s : String; VAR NewCount : integer);
PROCEDURE Failure(VAR NW : Network; s : String);
PROCEDURE NoCut(VAR NW : Network; s: String);
PROCEDURE CutFail(VAR NW : Network; s: String; VAR NewCount : integer);
PROCEDURE ProcessQ(VAR NW : Network);
PROCEDURE Cleanup(VAR NW : Network);
PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
            threshold, sigmaL : real);
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU, PE : real);
(* ------------------------------------------------------------------------
*)
IMPLEMENTATION
USES
    Crt, DOS, FileChck;
CONST
   EVENTMIN = 98;
   EVENTMAX = 39998;
   EPSILON = 1.0E-2;
VAR
    NW : Network;              {Network graph}
    origin, destination, hopmax : integer;
    NumNodes : integer;  {maximum node index used}
    OldQ, NewQ, OldQC, NewQC : Text;
    ReadStr, WriteStr, ReadStrC, WriteStrC : string;
    OQS, QueueSize, TotCount, SuccCount, FailCount : LongInt;
    OQSC, QSizeC, TotCCount, SuccCCount, FailCCount : LongInt;
    SuccProb, FailProb, SuccPrC, FailPrC, Estimate : real;
    Exitflag : Boolean;
(* ------------------------------------------------------------------------
*)
FUNCTION Pfunction(z : real) : real;
```

[For a listing of this function, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE SetParam(NW : Network);
VAR
  Event : string;
  Happen : boolean;
BEGIN
  TotCount := 0;
  TotCCount := 0;
  SuccCount := 0;
  SuccCCount := 0;
  FailCount := 0;
  FailCCount := 0;
  QueueSize := 1;
  QSizeC := 1;
  SuccProb := 0.0;
  SuccPrC := 0.0;
  FailProb := 0.0;
  FailPrC := 0.0;
  ReadStr := 'DATA2.TMP';
  ReadStrC := 'DATA4.TMP';
  WriteStr := 'DATA1.TMP';
```

```
  WriteStrC := 'DATA3.TMP';
  happen := CheckNewFile(NewQ, WriteStr);
  happen := CheckNewFile(NewQC, WriteStrC);
  happen := CheckNewFile(OldQ, ReadStr);
  happen := CheckNewFile(OldQC, ReadStrC);
  WITH NW DO
    BEGIN
      InitialEvent(Event, '1', EdgeNum);
      SaveEvent(NewQ, NW, Event);
      SaveEvent(NewQC, NW, Event);
      SwapFiles(OldQ, NewQ, OldQC, NewQC);
      SaveEvent(NewQ, NW, Event);
      SaveEvent(NewQC, NW, Event);
    END; (* WITH *)
END; (* SetParam *)
(* ------------------------------------------------------------------------
*)
PROCEDURE SwapFiles(VAR OldQ, NewQ, OldQC, NewQC : Text);
VAR
 s : string;
 happen : boolean;
BEGIN
  Close(NewQ);
  Close(NewQC);
  Close(OldQ);
  Close(OldQC);
  s := ReadStr;
  ReadStr := WriteStr;
  WriteStr := s;
  s := ReadStrC;
  ReadStrC := WriteStrC;
  WriteStrC := s;
  happen := CheckOldFile(OldQ, ReadStr);
  happen := CheckOldFile(OldQC, ReadStrC);
  happen := CheckNewFile(NewQ, WriteStr);
  happen := CheckNewFile(NewQC, WriteStrC);
END; (* SwapFiles *)
(* ------------------------------------------------------------------------
*)
PROCEDURE Success(VAR NW : Network; s : string; VAR NewCount : integer);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE NoCut(VAR NW : Network; s : String);
```

[For a listing of this procedure, see Section D.2.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE Failure(VAR NW : Network; s: string);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE CutFail(VAR NW : Network; s : String; VAR NewCount : Integer);
```

[For a listing of this procedure, see Section D.2.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE ProcessQ(VAR NW : Network);
VAR
  Event : String;
  PathLen, NewCount : Integer;
BEGIN
  OQS := QueueSize;
  OQSC := QSizeC;
  Exitflag := FALSE;
  QueueSize := 0;
  SwapFiles(OldQ, NewQ, OldQC, NewQC);
  WHILE NOT EOF(OldQ) DO
    BEGIN
      ReadEvent(OldQ, Event);
      SetEventLinks(NW, Event);
      IF TwoWaySearch(NW, PathLen) THEN
        BEGIN (* Path found *)
          Success(NW, Event, NewCount);
          ComplementEvent(NewQ, NW, Event, NewCount);
          Inc(QueueSize, NewCount);
        END; (* Path found *)
      RestoreEventLinks(NW);
      Estimate := 0.5*(1.0 + SuccProb - FailPrC);
      IF TotCount > EVENTMAX THEN Exit;
      IF ((1.0 - (SuccProb + FailPrC)) < EPSILON) AND
               (succcount+failccount>eventmin) THEN
        BEGIN
          Exitflag := TRUE;
          Exit;
        END;
    END; (* OldQ *)
   IF (QueueSize = 0) THEN Exit;
   QSizeC := 0;
   WHILE NOT EOF(OldQC) DO
    BEGIN
      ReadEvent(OldQC, Event);
      SetEventLinks(NW, Event);
      IF CutSetSearch(NW, Event) THEN
        BEGIN (* Cut Set found *)
          CutFail(NW, Event, NewCount);
          AddEvents(NewQC, NW, Event, NewCount);
          Inc(QSizeC, NewCount);
        END; (* Cut Set found *)
      RestoreEventLinks(NW);
      IF TotCCount > EVENTMAX THEN Exit;
      IF ((1.0 - (SuccProb + FailPrC)) < EPSILON)
               AND (succcount+failccount>eventmin) THEN
        BEGIN
         Exitflag := TRUE;
         Exit;
        END;
    END; (* OldQC *)
END; (* ProcessQ *)
(* ------------------------------------------------------------------------
*)
PROCEDURE Cleanup(VAR NW : Network);
BEGIN
  Close(NewQ);
  Close(OldQ);
```

```
  Erase(NewQ);
  Erase(OldQ);
  Close(NewQC);
  Close(OldQC);
  Erase(NewQC);
  Erase(OldQC);
END; (* Cleanup *)
(* ------------------------------------------------------------------
*)

PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
                  Threshold, SigmaL : real);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------
*)
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU, PE : real);
BEGIN
  WITH NW DO
  BEGIN
    source := orig;
    sink := dest;
    SetParam(NW);
    WHILE (QueueSize <> 0) AND (QSizeC <> 0) DO
      ProcessQ(NW);
    PL := SuccProb;
    PU := 1.0 - FailPrC;
    PE := 0.5*(PL + PU);
    IF (QueueSize = 0) AND NOT Exitflag THEN PE := PL
    ELSE IF (QSize = 0) AND NOT Exitflag THEN PE := PU;
    Cleanup(NW);
  END; (* WITH *)
END; (* ELReliabil *)
END.
```

D.4 IMPLEMENTATION OF ALGORITHM SELECTION

D.4.1 **Program ELCUTPAT.PAS**

```
PROGRAM ELCutPat;
USES
    Crt, Dos, CutPath, PopMenus, Dir_Menu, Strings, Keyboard, FileChck,
    NetSet;
VAR
   NW : Network;
   NumNodes, NumJams, kj, hopmax : integer;
   Thr, Sig : real;
(* ------------------------------------------------------------------ *)
PROCEDURE InitializeSINR(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE GetSNRs(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
```

```
PROCEDURE GetData;
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE Testbeta(VAR SINR : ThreeDarray);
```

[For a listing of this procedure, see Section D.1.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE TestSTRel(VAR NW : Network);
```

[For a listing of this procedure, see Section D.2.1]

```
(* ---------------------- Main Program ----------------------------- *)
BEGIN
  ClrScr;
  GetSNRs(SINR);
  GetData;
  Testbeta(SINR);
  OpenNetwork(NW);
  TestSTRel(NW);
  CloseNetwork(NW);
END. (* Main Program *)
```

D.4.2 Unit CUTPATH.PAS

```
UNIT CUTPATH;
INTERFACE
USES
    NetSet;
CONST
    maxv = 20;          {Maximum number of vertices in the graph}
    maxe = 135;          {Maximum number of edges}
    maxjam = 10;           {Maximum number of jammers}
TYPE
    Vectj = array[1..maxjam] of real;
    Matrxi = array[1..maxv] of Vectj;
    ThreeDarray = array[1..maxv] of Matrxi;
VAR
    SINR, BetaAll : ThreeDarray;
    Threshold, SigmaL : Real;
    Orig, Dest : integer;
    Fname : string;
    Fileout : text;
FUNCTION Pfunction(z : real) : real;
PROCEDURE SwapFiles(VAR OldQ, NewQ : Text);
PROCEDURE SetParam(NW : Network);
PROCEDURE Success(VAR NW : Network; s : String; VAR NewCount : integer);
PROCEDURE Failure(VAR NW : Network; s : String);
PROCEDURE NoCut(VAR NW : Network; s: String);
PROCEDURE CutFail(VAR NW : Network; s: String; VAR NewCount : integer);
PROCEDURE ProcessQ(VAR NW : Network);
PROCEDURE Cleanup(VAR NW : Network);
PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
            threshold, sigmaL : real);
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU, PE : real);
(* ------------------------------------------------------------------
*)
IMPLEMENTATION
```

```
USES
    Crt, DOS, FileChck;
CONST
   EVENTMIN = 98;
   EVENTMAX = 39998;
   EPSILON = 1.0E-2;
VAR
    NW : Network;              {Network graph}
    origin, destination, hopmax : integer;
    NumNodes : integer;  {maximum node index used}
    OldQ, NewQ : Text;
    ReadStr, WriteStr : string;
    QueueSize, TotCount, SuccCount, FailCount : LongInt;
    SuccProb, FailProb, Estimate : real;
(* ------------------------------------------------------------------------
*)
FUNCTION Pfunction(z : real) : real;
```

[For a listing of this function, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE SetParam(NW : Network);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE SwapFiles(VAR OldQ, NewQ : Text);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE Success(VAR NW : Network; s : string; VAR NewCount : integer);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE GetNew(VAR NW : Network; s : string; VAR NewCount : integer);
VAR
  i : integer;
BEGIN
  WITH NW DO
  BEGIN
    NewCount := 0;
    FOR i := 1 TO EdgeNum DO
      IF (s[i] = '1') AND (UpEdges[i] <> '1') THEN
        Inc(NewCount);
  END; (* WITH *)
END; (* GetNew *)
(* ------------------------------------------------------------------------
*)
PROCEDURE NoCut(VAR NW : Network; s : String);
```

[For a listing of this procedure, see Section D.2.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE Failure(VAR NW : Network; s: string);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE CutFail(VAR NW : Network; s : String; VAR NewCount : Integer);
```

[For a listing of this procedure, see Section D.2.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE ProcessQ(VAR NW : Network);
VAR
  Event, EdgesUp : String;
  PathLen, NewCount, CutCount : Integer;
  ListPath : EdgeList;
  happen : Boolean;
  i : integer;
LABEL
  DONE;
BEGIN
  QueueSize := 0;
  SwapFiles(OldQ, NewQ);
  WHILE NOT EOF(OldQ) DO
  BEGIN
    ReadEvent(OldQ, Event);
    SetEventLinks(NW, Event);
    IF NOT TwoWaySearch(NW, PathLen) THEN
      BEGIN (* Definite failure *)
        Failure(NW, Event);
        GOTO DONE;
      END; (* Definite failure *)
    GetNew(NW, Event, NewCount);
    IF (NewCount < 2) THEN
      BEGIN (* Definite success or short addition *)
        Success(NW, Event, NewCount);
        IF (NewCount > 0) THEN
          ComplementEvent(NewQ, NW, Event, NewCount);
        Inc(QueueSize, NewCount);
        GOTO DONE;
      END; (* Definite success or short addition *)
    EdgesUp := NW.UpEdges;
    FOR i := 1 TO NW.EdgeNum DO
      ListPath[i] := NW.PathList^[i];
    happen := CutSetSearch(NW, Event);
    GetNew(NW, Event, CutCount);
    IF (CutCount < NewCount) THEN
      BEGIN (* Cutset preferred *)
        CutFail(NW, Event, CutCount);
        AddEvents(NewQ, NW, Event, CutCount);
        Inc(QueueSize, CutCount);
      END (* Cutset preferred *)
    ELSE
      BEGIN (* Path preferred *)
        NW.UpEdges := EdgesUp;
        FOR i := 1 TO NW.EdgeNum DO
          NW.PathList^[i] := ListPath[i];
        Success(NW, Event, NewCount);
        ComplementEvent(NewQ, NW, Event, NewCount);
        Inc(QueueSize, NewCount);
      END; (* Path preferred *)
DONE:
    RestoreEventLinks(NW);
```

```
    Estimate := 0.5*(1.0 + SuccProb - FailProb);
    IF TotCount > EVENTMAX THEN Exit;
    IF ((1.0 - (SuccProb + FailProb)) < EPSILON) AND
              (TotCount > EVENTMIN) THEN Exit;
  END; (* WHILE *)
END; (* ProcessQ *)
(* ------------------------------------------------------------------------
*)
PROCEDURE Cleanup(VAR NW : Network);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------- *)
PROCEDURE GetNett(VAR NW : Network; NumNodes, kj, hopmax : integer;
                  Threshold, SigmaL : real);
```

[For a listing of this procedure, see Section D.1.2]

```
(* ------------------------------------------------------------------------
*)
PROCEDURE ELReliabil(VAR NW : Network; VAR PL, PU, PE : real);
```

[For a listing of this procedure, see Section D.2.2]

```
END.
```

D.5 UNIT NETSET.PAS

Note that many of the procedures and functions in this unit are not used by the programs studied in this report.

```
UNIT NETSET;
(*  Dotson unit  NETSET.PAS     (includes cutsets w/o path search)
      This unit defines the structure of a network in terms of the
   adjacency matrix and its edge probability vector.  In addition,
   various edge vectors and also a path string are included in the
   structure, principally to facilitate the operation of  Dotson's
   algorithm.  Other elements are the number of vertices and edges
   as well as the source and sink nodes, and the hop maximum.  Two
   path search routines are provided, a forward only and a two-way
   search procedure.  The data file formats used are implicit here,
   as described in an earlier memorandum.
      EdgeFile  - Input:  description of network configuration
      EventFile - Output: collection of success/failure events
      NewQueue, OldQueue - Temporary:  files for event storage
   The calling programs must control file management for this data.  *)
INTERFACE
CONST
  OK           = 0;
  NODEMAX      = 40;
  EDGEMAX      = 255;
  FORMAT_ERROR = 1;
  DATA_ERROR   = 2;
  INDEX_ERROR  = 3;
TYPE
  NodeList = ARRAY [1..NODEMAX] OF Byte;
  NodeVect = ARRAY [1..NODEMAX] OF Real;
  NodeMat  = Array [1..NODEMAX] OF NodeVect;
  EdgeList = ARRAY [1..EDGEMAX] OF Byte;
  EdgeVect = ARRAY [1..EDGEMAX] OF Real;
  EdgeMat  = Array [1..NODEMAX] OF EdgeVect;
```

```
  AdjcMat  = ARRAY [1..NODEMAX] OF NodeList;
  PathMat  = ARRAY [0..NODEMAX + 1] OF NodeList;
  NodeListPtr = ^NodeList;
  NodeVectPtr = ^NodeVect;
  NodeMatPtr  = ^NodeMat;
  EdgeListPtr = ^EdgeList;
  EdgeVectPtr = ^EdgeVect;
  EdgeMatPtr  = ^EdgeMat;
  AdjcMatPtr  = ^AdjcMat;
  LinkSet = SET OF 1..EdgeMax;
  NodeSet = SET OF 1..NodeMax;
  Network  = RECORD
    Source   : Integer;
    Sink     : Integer;
    NodeNum  : Integer;
    EdgeNum  : Integer;
    MaxHops  : Integer;
    GraphMat : AdjcMatPtr;
    Beta     : EdgeVectPtr;
    Alpha    : NodeVectPtr;
    PathList : EdgeListPtr;
    I_Index  : EdgeListPtr;
    J_Index  : EdgeListPtr;
    UpEdges  : String;
    EdgeStr  : String;
    EdgeFile : Text;
  END; (* Network - basic network structure for Dotson method *)
TYPE
  NodeData = RECORD
    EdgeMat   : AdjcMatPtr;
    EdgesOn   : EdgeListPtr;
    EdgesOff  : EdgeListPtr;
    Gamma     : EdgeVectPtr;
    MaxDegree : Integer;
    NumOn     : Integer;
    NumOff    : Integer;
  END; (* NodeData - node structure for equivalent links method *)
(*-------------------------------------------------------------------
---------   PUBLIC FUNCTIONS AND PROCEDURES -------------------------
-------------------------------------------------------------------*)
PROCEDURE   OpenNetwork(VAR NW : Network);
PROCEDURE   CloseNetwork(VAR NW : Network);
PROCEDURE   SetNetworkNodes(VAR NW : Network);
PROCEDURE   RestoreEventLinks(VAR NW : Network);
FUNCTION    ReadEdgeFile(VAR NW : Network) : Integer;
FUNCTION    ProbEvent(VAR NW    : Network;
                          Event : String) : Real;
PROCEDURE   SetEventLinks(VAR NW    : Network;
                              Event : String);
PROCEDURE   ReadEvent(VAR OldQueue : Text;
                      VAR Event    : String);
PROCEDURE   InitialEvent(VAR Event : String;
                             Value : Char;
                             n     : Integer);
PROCEDURE   SaveEvent(VAR NewQueue : Text;
                      VAR NW       : Network;
                          Event    : String);
FUNCTION    OneWaySearch(VAR NW : Network;
                         VAR PathLen : Integer) : Boolean;
FUNCTION    TwoWaySearch(VAR NW      : Network;
```

```
                          VAR PathLen : Integer) : Boolean;
PROCEDURE   ComplementEvent(VAR NewQueue : Text;
                            VAR NW       : Network;
                                Event    : String;
                                Count    : Integer);
FUNCTION    SuccessEvent(VAR EventFile : Text;
                          VAR NW        : Network;
                              Event     : String;
                          VAR NewCount  : Integer;
                              Flag      : Integer) : Real;
FUNCTION    FailureEvent(VAR EventFile : Text;
                          VAR NW        : Network;
                              Event     : String;
                              Flag      : Integer) : Real;
FUNCTION    PathCheck(NW : Network; Event : String) : Boolean;
FUNCTION    CutSetSearch(VAR NW : Network; Event : String) : Boolean;
PROCEDURE   AddEvents(VAR NewQC : Text; VAR NW : Network;
                    Event : String; NewCount : Integer);
PROCEDURE   CloseNodeData(VAR ND : NodeData);
PROCEDURE   OpenNodeData(VAR NW : Network;
                          VAR ND : NodeData);
FUNCTION    BackFitNodes(VAR NW    : Network;
                          VAR ND    : NodeData;
                              Event : String) : Real;
PROCEDURE   EdgeFileError(s : String);
PROCEDURE   EventFileError(s : String);
PROCEDURE   AlphaFileError(s : String);
PROCEDURE   DataIndexError(s0, s1 : String);
PROCEDURE   DataConflictError(VAR NW   : Network;
                                  s    : String;
                                  m, n : Integer);
(*-------------------------------------------------------------------*)
IMPLEMENTATION
USES
  Crt, Keyboard, Strings, TextScrn, PopMenus;
(*-------------------------------------------------------------------
---------   FUNCTIONS AND PROCEDURES PRIVATE TO THIS UNIT  -----------
-------------------------------------------------------------------*)
PROCEDURE   SetAdjacencyMat(MatPtr : AdjcMatPtr);
FUNCTION    CheckEdgeFileData(NW : Network) : Boolean;
PROCEDURE   SetLinkMat(VAR NW : Network;
                       VAR pm : PathMat);
PROCEDURE   SetNodeList(VAR nl    : NodeList;
                            Value : Byte);
PROCEDURE   SetEdgeList(VAR ElPtr : EdgeListPtr;
                            Value : Byte);
PROCEDURE   SetEdgeVect(ProbPtr : EdgeVectPtr;
                        Value   : Real);
PROCEDURE   SetNodeVect(ProbPtr : NodeVectPtr;
                        Value   : Real);
PROCEDURE   WriteEvent(VAR EventFile : Text;
                           NW        : Network;
                         Event     : String;
                         Flag      : Integer);
FUNCTION    OneWayPath(VAR NW      : Network;
                       VAR pm      : PathMat;
                       VAR PathLen : Integer) : Boolean;
FUNCTION    ForwardSearch(VAR NW : Network;
                          VAR pm : PathMat;
                          VAR kf : Integer
```

```
                          VAR FCount: Integer) : Boolean;
FUNCTION    BackwardSearch(VAR NW : Network;
                           VAR pm : PathMat;
                           VAR kr : Integer
                           VAR BCount: Integer) : Boolean;
FUNCTION    TwoWayPath(VAR NW : Network;
                       VAR pm : PathMat;
                       VAR pl, rn, kf, kr : Integer) : Boolean;
PROCEDURE   SetWorkingEdges(    Event : String;
                            VAR ND    : NodeData);
FUNCTION    BackFitProb(    Up   : EdgeList;
                            Down : EdgeList;
                        VAR ND   : NodeData;
                            Prob : Real)     : Real;
----------------------------------------------------------------------*)

(*----------------------------------------------------------------------*)
PROCEDURE OpenNetwork(VAR NW : Network);
(* Set up network pointers on the heap - no initialization *)
BEGIN
  WITH NW DO
    BEGIN
      New(GraphMat);
      New(I_Index);
      New(J_Index);
      New(PathList);
      New(Alpha);
      New(Beta);
    END; (* WITH *)
END; (* OpenNetwork *)
(*----------------------------------------------------------------------
*)
PROCEDURE CloseNetwork(VAR NW : Network);
(* Restore network pointers to the pre-network heap status *)
BEGIN
  WITH NW DO
    BEGIN
      Dispose(Beta);
      Dispose(Alpha);
      Dispose(PathList);
      Dispose(J_Index);
      Dispose(I_Index);
      Dispose(GraphMat);
    END; (* WITH *)
END; (* CloseNetwork *)
(*----------------------------------------------------------------------
*)
PROCEDURE SaveEvent(VAR NewQueue: Text;
                    VAR NW      : Network;
                        Event   : String);
(* Save the edge event to the temporary file - i.e., to the next queue
*)
BEGIN
  Event[0] := Chr(NW.EdgeNum);
  Writeln(NewQueue, Event);
END; (* SaveEvent *)
(*----------------------------------------------------------------------
*)
PROCEDURE ReadEvent(VAR OldQueue : Text;
                    VAR Event    : String);
```

```
(* Read an edge event from the temporary file - i.e., the current queue
*)
BEGIN
  Readln(OldQueue, Event);
  DropLeadingBlanks(Event);
END; (* ReadEvent *)
(*-------------------------------------------------------------------
*)
PROCEDURE WriteEvent(VAR EventFile : Text;
                         NW        : Network;
                         Event     : String;
                         Flag      : Integer);
(* Write success or failure edge event with the associated integer flag
*)
BEGIN
  Event[0] := Chr(NW.EdgeNum);
  Write(EventFile, Flag : 3, ' ');
  Writeln(EventFile, Event);
END; (* WriteEvent *)
(*-------------------------------------------------------------------
*)
FUNCTION CheckEdgeFileData(NW : Network) : Boolean;
(* Check EdgeFile data against maximum allowable size - show any error
*)
BEGIN
  CheckEdgeFileData := TRUE;
  IF (NW.NodeNum > NODEMAX) OR (NW.EdgeNum > EDGEMAX) THEN
    BEGIN
      CursorOff;
      CheckEdgeFileData := FALSE;
      QuickPopUp(10, 4, 70, 9, 2, Yellow, Red, '');
      Writeln('  ERROR');
      Writeln(Bell);
      IF NW.NodeNum > NODEMAX THEN
            Writeln('        Maximum number of nodes (', NODEMAX, ')
exceeded');
      IF NW.EdgeNum > EDGEMAX THEN
            Writeln('        Maximum number of edges (', EDGEMAX, ')
exceeded');
      Writeln;
      Pause('  Press any key to continue... ');
      ClosePopUp;
      CursorOn;
      Exit;
    END;
END; (* CheckEdgeFileData *)
(*-------------------------------------------------------------------*)
PROCEDURE SetEdgeList(VAR ElPtr : EdgeListPtr;
                          Value : Byte);
(* Initialize an EdgeListPtr - all of its entries set to given value *)
VAR
  i : Integer;
BEGIN
  FOR i := 1 TO EDGEMAX DO
    ElPtr^[i] := Value;
END; (* SetEdgeList *)
(*-------------------------------------------------------------------*)
PROCEDURE SetNodeList(VAR nl    : NodeList;
                          Value : Byte);
(* Initialize an NodeList - all of its entries are set to given value *)
```

```
VAR
  i : Integer;
BEGIN
  FOR i := 1 TO NODEMAX DO
    nl[i] := Value;
END; (* SetNodeList *)
(*-------------------------------------------------------------------------
*)
PROCEDURE SetAdjacencyMat(MatPtr : AdjcMatPtr);
(* Initialize network adjacency matrix - all its entries equal to zero
*)
VAR
  i : Integer;
BEGIN
  FOR i := 1 TO NODEMAX DO
    SetNodeList(MatPtr^[i], 0);
END; (* SetAdjacencyMat *)
(*-------------------------------------------------------------------------
*)
PROCEDURE SetEdgeVect(ProbPtr : EdgeVectPtr;
                      Value   : Real);
(* Initialize edge vector - set all edge probabilities to given value *)
VAR
  i : Integer;
BEGIN
  FOR i := 1 TO EDGEMAX DO
    ProbPtr^[i] := Value;
END; (* SetEdgeVect *)
(*-------------------------------------------------------------------------
*)
PROCEDURE SetNodeVect(ProbPtr : NodeVectPtr;
                      Value   : Real);
(* Initialize node vector - set all node probabilities to given value *)
VAR
  i : Integer;
BEGIN
  FOR i := 1 TO NODEMAX DO
    ProbPtr^[i] := Value;
END; (* SetNodeVect *)
(*-------------------------------------------------------------------------
*)
FUNCTION ProbEvent(VAR NW    : Network;
                       Event : String)   : Real;
(* Use edge probabilities, return probability of the edge string event
*)
VAR
  Prob : Real;
  i    : Integer;
BEGIN
  WITH NW DO
    BEGIN
      Prob := 1.0;
      FOR i := 1 TO EdgeNum DO
        BEGIN
          IF Event[i] = '2' THEN
            Prob := Prob * Beta^[i]
          ELSE IF Event[i] = '0' THEN
            Prob := Prob * (1.0 - Beta^[i]);
        END;
    END; (* WITH *)
```

```
  ProbEvent := Prob
END; (* ProbEvent *)
(*-------------------------------------------------------------------------
*)
PROCEDURE InitialEvent(VAR Event : String;
                           Value : Char;
                           n     : Integer);
(* Initialize event string of length n - all entries equal given value
*)
BEGIN
  FillChar(Event, Sizeof(String), Value);
  Event[0] := Chr(n);
END; (* InitialEvent *)
(*-------------------------------------------------------------------------
*)
PROCEDURE SetNetworkNodes(VAR NW : Network);
(* Get source and sink nodes and hop number for network from keyboard *)
VAR
  i : Integer;
BEGIN
  WITH NW DO
    BEGIN
      Writeln('  ', EdgeStr);
      Writeln(' Make Node Selections: ');
      Write(' Source Node ? ________ ');
      IF(0 = StringToInt(PosIntegerInp(''), i)) THEN
        Source := i
      ELSE
        Source := 1;
      IF (Source < 1) OR (Source > NodeNum) THEN
        Source := 1;
      Writeln;
      Write(' Terminal Node ? ______ ');
      IF (0 = StringToInt(PosIntegerInp(''), i)) THEN
        Sink := i
      ELSE
        Sink := NodeNum;
      IF (Sink < 1) OR (Sink > NodeNum) THEN
        Sink := NodeNum;
      Writeln;
      Write(' Hop Maximum ? ________ ');
      IF (0 = StringToInt(PosIntegerInp(''), i)) THEN
        MaxHops := i
      ELSE
        MaxHops := NodeNum - 1;
      Writeln;
      IF (MaxHops < 1) OR (MaxHops >= NodeNum) THEN
        MaxHops := NodeNum - 1;
      IF Source = Sink THEN
        BEGIN
          CursorOff;
          QuickPopUp(20, 16, 70, 17, 2, Yellow, Red,
                     ' NOTE - Source and sink are not distinct ');
          Writeln(#7);
          Pause(' Press any key to continue...');
          MaxHops := 0;
          CursorOn;
          ClosePopUp;
        END;
    END; (* WITH *)
```

```
END; (* SetNetworkNodes *)
(*-------------------------------------------------------------------------
*)
PROCEDURE SetEventLinks(VAR NW    : Network;
                            Event : String);
(* Turn off links according to failed connections in edge string Event
*)
VAR
  i : Integer;
BEGIN
  WITH NW DO
    BEGIN
      InitialEvent(UpEdges, '1', EdgeNum);
      SetEdgeList(PathList, 0);
      FOR i := 1 TO EdgeNum DO
        IF Event[i] = '0' THEN
          GraphMat^[I_Index^[i], J_Index^[i]] := 0;
      FOR i := 1 TO NodeNum DO
        BEGIN
          GraphMat^[i, source] := 0;
          GraphMat^[sink, i] := 0;
        END;
    END; (* WITH *)
END; (* SetEventLinks *)
(*-------------------------------------------------------------------------*)
PROCEDURE RestoreEventLinks(VAR NW : Network);
(* Restore all links for the original adjacency matrix of the network *)
VAR
  k : Integer;
BEGIN
  WITH NW DO
    BEGIN
      FOR k := 1 TO EdgeNum DO
        GraphMat^[I_Index^[k], J_Index^[k]] := k;
      FOR k := 1 TO NodeNum DO
        GraphMat^[k,k] := 0;
    END; (* WITH *)
END; (* RestoreEventLinks *)
(*-------------------------------------------------------------------------*)
PROCEDURE ComplementEvent(VAR NewQueue : Text;
                          VAR NW       : Network;
                              Event    : String;
                              Count    : Integer);
(* Get complementary edge events and save them to the next queue file *)
VAR
  i, j, k   : Integer;
  CompEvent :  String;
BEGIN
  WITH NW DO
    BEGIN
      j := 0;
      FOR k := 1 TO Count DO
        BEGIN
          CompEvent := Event;
          REPEAT
            Inc(j);
          UNTIL (Event[PathList^[j]] <> '2');
          FOR i := 1 TO j - 1 DO
            CompEvent[PathList^[i]] := '2';
          CompEvent[PathList^[j]] := '0';
```

```
          SaveEvent(NewQueue, NW, CompEvent);
        END;
    END; (* WITH *)
END; (* ComplementEvent *)
(*-------------------------------------------------------------------------
*)
PROCEDURE AddEvents(VAR NewQC : Text; VAR NW : Network; Event : String;
                    NewCount : Integer);
VAR
  j, k : Integer;
  CompEvent : String;
BEGIN
  WITH NW DO
  BEGIN
    FOR k := 1 TO NewCount DO
      BEGIN
        CompEvent := Event;
        CompEvent[PathList^[k]] := '2';
        IF k>1 THEN
          FOR j := 1 TO k-1 DO
            CompEvent[PathList^[j]] := '0';
        SaveEvent(NewQC, NW, CompEvent);
      END; (* FOR *)
  END; (* WITH *)
END; (* Add Events *)
(*-------------------------------------------------------------------------
*)
FUNCTION SuccessEvent(VAR EventFile : Text;
                      VAR NW        : Network;
                          Event     : String;
                      VAR NewCount  : Integer;
                          Flag      : Integer) : Real;
(* Process the success event and write it to the output event file  *)
(* This event has positive length - return probability of the event *)
VAR
  SuccEvent : String;
  i         : Integer;
BEGIN
  WITH NW DO
    BEGIN
      NewCount := 0;
      SuccEvent[0] := Char(EdgeNum);
      FOR i := 1 TO EdgeNum DO
        BEGIN
          IF UpEdges[i] = '1' THEN
            SuccEvent[i] := Event[i]
          ELSE
            SuccEvent[i] := UpEdges[i];
          IF (Event[i] = '1') AND (UpEdges[i] <> '1') THEN
            Inc(NewCount);
        END;
    END; (* WITH *)
  WriteEvent(EventFile, NW, SuccEvent, Flag);
  SuccessEvent := ProbEvent(NW, SuccEvent);
END; (* SuccessEvent *)
(*-------------------------------------------------------------------------
*)
FUNCTION FailureEvent(VAR EventFile : Text;
                      VAR NW        : Network;
                          Event     : String;
```

```
                          Flag      : Integer) : Real;
(* Process the failure event and write it to the output event file  *)
(* This event has negative length - return probability of the event *)
BEGIN
  WriteEvent(EventFile, NW, Event, -Flag);
  FailureEvent := ProbEvent(NW, Event);
END; (* FailureEvent *)
(*-------------------------------------------------------------------*)
FUNCTION ReadEdgeFile(VAR NW : Network) : Integer;
(* Read EdgeFile data and build the adjacency matrix for the network *)
LABEL
  GRAPH_ERROR;
VAR
  t       : Real;
  i, j, k : Integer;
BEGIN
  ReadEdgeFile := OK;
  WITH NW DO
    BEGIN
      {$I-} Readln(EdgeFile, Source, Sink, NodeNum, EdgeNum); {$I+}
      IF (IOResult <> 0) OR (ExitCode <> 0) THEN
        GoTo GRAPH_ERROR;
      IF NOT CheckEdgeFileData(NW) THEN
        BEGIN
          ReadEdgeFile := DATA_ERROR;
          Exit;
        END;
      SetAdjacencyMat(GraphMat);
      FOR k := 1 TO EdgeNum DO
        BEGIN
          {$I-} Readln(EdgeFile, i, j, t); {$I+}
          IF (IOResult <> 0) OR (ExitCode <> 0) THEN
            GoTo GRAPH_ERROR;
          I_Index^[k] := i;
          J_Index^[k] := j;
          GraphMat^[i,j] := k;
          Beta^[k] := t;
        END;
    END; (* WITH *)
  Exit;
  GRAPH_ERROR:
    ReadEdgeFile := FORMAT_ERROR;
END; (* ReadEdgeFile *)
(*-------------------------------------------------------------------
-*)
FUNCTION OneWayPath(VAR NW      : Network;
                    VAR pm      : PathMat;
                    VAR PathLen : Integer) : Boolean;
(* Is there a path from Source to Sink using current adjacency matrix ?
*)
(* Use forward flood searches to find path - no backward search is used
*)
VAR
  i, j, k   : Integer;
  Path, Chk : Boolean;
BEGIN
  WITH NW DO
    BEGIN
      SetNodeList(pm[NodeNum+1], 0);
      Move(pm[NodeNum+1], pm[0], NodeNum*SizeOf(Byte));
```

```
      pm[1] := GraphMat^[Source];
      pm[0, Source] := 1;
      Path := FALSE;
      OneWayPath := FALSE;
      FOR k := 1 TO MaxHops DO
        BEGIN  (* Main Loop *)
          Chk := FALSE;
          Move(pm[NodeNum+1], pm[k+1], NodeNum*SizeOf(Byte));
          FOR i := 1 TO NodeNum DO
            BEGIN
              IF pm[k,i] > 0 THEN
                BEGIN
                  Chk := TRUE;
                  IF i = Sink THEN
                    BEGIN
                      OneWayPath := TRUE;
                      Path := TRUE;
                      PathLen := k;
                      Exit;
                    END;
                  FOR j := 1 TO NodeNum DO
                      IF (GraphMat^[i,j]>0) AND NOT (pm[0,j]> 0) THEN
                        BEGIN
                          pm[k + 1,j] := GraphMat^[i,j];
                          pm[0,j] := 1;
                        END;
                END; (* IF *)
            END; (* i-Loop *)
          IF Path THEN
            Exit;
          IF NOT Chk THEN
            BEGIN
              PathLen := k - 1;
              Exit;
            END;
        END; (* k-Loop *)
      PathLen := k;
    END; (* WITH *)
END; (* OneWayPath *)
(*-------------------------------------------------------------------------
*)
FUNCTION OneWaySearch(VAR NW      : Network;
                      VAR PathLen : Integer) : Boolean;
(* If an st-path is found, set up the path links used and return  TRUE
*)
VAR
  LinkMat         : PathMat;
  k, Nback , Link : Integer;
BEGIN
  WITH NW DO
    BEGIN
      OneWaySearch := FALSE;
      IF NOT OneWayPath(NW, LinkMat, PathLen) THEN
        Exit;
      InitialEvent(UpEdges, '1', EdgeNum);
      SetEdgeList(PathList, 0);
      Nback := Sink;
      FOR k := 1 TO PathLen DO
        BEGIN
          Link := LinkMat[PathLen - k + 1, Nback];
```

```
          UpEdges[Link] := '2';
          PathList^[PathLen - k + 1] := Link;
          Nback := I_Index^[Link];
        END;
      OneWaySearch := TRUE;
    END; (* WITH *)
END; (* OneWaySearch *)
(*-------------------------------------------------------------------*)
PROCEDURE SetLinkMat(VAR NW : Network;
                     VAR pm : PathMat);
(* Initialize the link matrix before beginning a two_way path search *)
VAR
  i : Integer;
BEGIN
  WITH NW DO
    BEGIN
      FOR i := 0 TO NodeNum + 1 DO
        SetNodeList(pm[i], 0);
      pm[0, Source] := 1;
      pm[NodeNum + 1, Sink] := 1;
      pm[1] := GraphMat^[Source];
      FOR i := 1 TO NodeNum DO
        pm[NodeNum, i] := GraphMat^[i, Sink];
    END; (* WITH *)
END; (* SetLinkMat *)
(*-------------------------------------------------------------------
*)
FUNCTION ForwardSearch(VAR NW : Network;
                       VAR pm : PathMat;
                       VAR kf : Integer
                       VAR FCount : Integer) : Boolean;
(* Extend the path search one hop forward - i.e., away from the source
*)
VAR
  i, j, Count : Integer;
BEGIN
  WITH NW DO
    BEGIN
      Count := FCount;
      FOR i := 1 TO NodeNum DO
        IF pm[kf,i] > 0 THEN
            FOR j := 1 TO NodeNum DO
              IF (GraphMat^[i,j]>0) AND NOT (pm[0,j]>0) THEN
                BEGIN
                  pm[kf + 1, j] := GraphMat^[i,j];
                  pm[0, j] := 1;
                  Inc(FCount);
                END;
      IF Count <> FCount THEN ForwardSearch := TRUE
      ELSE ForwardSearch := FALSE;
      Inc(kf);
    END; (* WITH *)
END; (* ForwardSearch *)
(*-------------------------------------------------------------------*)
FUNCTION BackwardSearch(VAR NW : Network;
                        VAR pm : PathMat;
                        VAR kr : Integer
                        VAR BCount: Integer) : Boolean;
(* Extend the path search one hop backward - i.e., away from the sink *)
VAR
```

```
  i, j, Count : Integer;
BEGIN
  WITH NW DO
    BEGIN
      Count := BCount;
      FOR i := 1 TO NodeNum DO
        IF pm[NodeNum + 1 - kr, i] > 0 THEN
          FOR j := 1 TO NodeNum DO
            IF (GraphMat^[j,i]>0) AND NOT (pm[NodeNum+1, j]>0) THEN
              BEGIN
                pm[NodeNum - kr, j] := GraphMat^[j,i];
                pm[NodeNum + 1, j] := 1;
                Inc(BCount);
              END;
      Inc(kr);
      IF (Count <> BCount) THEN BackwardSearch := TRUE
      ELSE BackwardSearch := FALSE;
    END; (* WITH *)
END; (* BackwardSearch *)
(*-------------------------------------------------------------------------
-*)
FUNCTION TwoWayPath(VAR NW             : Network;
                    VAR pm             : PathMat;
                    VAR pl, rn, kf, kr : Integer) : Boolean;
(* Is there an st-path using the current network adjacency matrix ? *)
(* Use alterating forward and backward flood searches to find path  *)
VAR
  j, BCount, FCount : Integer;
  Toggle, Check : Boolean;
BEGIN
  WITH NW DO
    BEGIN
      kf := 1;
      kr := 1;
      FCount := 1;
      BCount := 1;
      FOR j := 1 TO NodeNum DO
        BEGIN
          IF (pm[1,j] > 0) THEN Inc(FCount);
          IF (pm[NodeNum,j] > 0) THEN Inc(BCount);
        END;
      Toggle := FALSE;
      TwoWayPath := FALSE;
      Check := FALSE;
      REPEAT
        pl := kf + kr;
        FOR j := 1 TO NodeNum DO
          BEGIN
            IF (pm[kf, j] > 0) AND (pm[NodeNum + 1 - kr, j] > 0) THEN
              BEGIN
                rn := j;
                TwoWayPath := TRUE;
                Exit;
              END;
          END;
        Toggle := NOT Toggle; { alternate forward and  backward step }
        IF Toggle THEN
          Check := ForwardSearch(NW, pm, kf, FCount)
        ELSE
          Check := BackwardSearch(NW, pm, kr, BCount);
```

```
      UNTIL (NOT Check) OR (kf+kr > MaxHops);
    END; (* WITH *)
END; (* TwoWayPath *)
(*-------------------------------------------------------------------------
*)
FUNCTION TwoWaySearch(VAR NW      : Network;
                      VAR PathLen : Integer) : Boolean;
(* If an st-path is found, set up the path links used and return  TRUE
*)
VAR
  LinkMat             : PathMat;
  Nfor, Nback, Link   : Integer;
  j, k, kf, kr, RowNo : Integer;
BEGIN
  WITH NW DO
    BEGIN
      Link := GraphMat^[Source, Sink];
      IF Link > 0 THEN
        BEGIN
          PathLen := 1;
          PathList^[1] := Link;
          UpEdges[Link] := '2';
          TwoWaySearch := TRUE;
          Exit;
        END;
      TwoWaySearch := FALSE;
      SetLinkMat(NW, LinkMat);
      IF NOT TwoWayPath(NW, LinkMat, PathLen, RowNo, kf, kr) THEN
        Exit;
      InitialEvent(UpEdges, '1', EdgeNum);
      SetEdgeList(PathList, 0);
      Nback := RowNo;
      FOR k := 1 TO kf DO
        BEGIN
          Link := LinkMat[kf - k + 1, Nback];
          UpEdges[Link] := '2';
          PathList^[kf - k + 1] := Link;
          Nback := I_Index^[Link];
        END;
      Nfor := RowNo;
      FOR k := 1 TO kr DO
        BEGIN
          Link := LinkMat[NodeNum - kr + k, Nfor];
          UpEdges[Link] := '2';
          PathList^[kf + k] := Link;
          Nfor := J_Index^[Link];
        END;
      TwoWaySearch := TRUE;
  END; (* WITH *)
END; (* TwoWaySearch *)
(*-------------------------------------------------------------------------*)
PROCEDURE AddSetF(NW : Network; VAR Links : PathMat; k : integer;
            VAR EventSet : LinkSet; VAR Cut : LinkSet;
            VAR Check : Boolean);
VAR
  i, j : integer;
BEGIN
  WITH NW DO
  BEGIN
    Check := FALSE;
```

```
    FOR i := 1 TO NodeNum DO (* sending node *)
      IF (Links[k-1,i] > 0) THEN  (* entry in event *)
        FOR j := 1 TO NodeNum DO  (* receiving node *)
          IF (GraphMat^[i,j] IN EventSet) AND NOT (Links[0,j]>0) THEN
            BEGIN  (* Propagate *)
              IF Links[k,j] = 0 THEN
                BEGIN
                  Links[k,j] := GraphMat^[i,j];
                  Check := TRUE;
                  Links[0,j] := 1;
                END;
            END (* Propagate *)
          ELSE IF (GraphMat^[i,j]>0) AND NOT (Links[0,j]>0) THEN
              Cut := Cut + [GraphMat^[i,j]];
  END; (* WITH *)
END; (* AddSetF *)
(*-------------------------------------------------------------------*)
PROCEDURE AddSetR(NW : Network; VAR Links : PathMat; k : integer;
            VAR EventSet : LinkSet; VAR Cut : LinkSet;
            VAR Check : Boolean);
VAR
  i, j : integer;
BEGIN
  WITH NW DO
  BEGIN
    Check := FALSE;
    FOR i := 1 TO NodeNum DO (* receiving node *)
      IF (Links[k+1,i] > 0) THEN (* entry in event *)
        FOR j := 1 TO NodeNum DO  (* sending node *)
          IF (GraphMat^[j,i] IN EventSet)
                AND NOT (Links[NodeNum+1,j]>0) THEN
            BEGIN  (* Propagate *)
              IF Links[k,j] = 0 THEN
                BEGIN
                  Links[k,j] := GraphMat^[j,i];
                  Check := TRUE;
                  Links[NodeNum+1,j] := 1;
                END;
            END (* Propagate *)
          ELSE IF (GraphMat^[j,i]>0) AND NOT (Links[NodeNum+1,j]>0) THEN
              Cut := Cut + [GraphMat^[j,i]];
  END; (* WITH *)
END; (* AddSetR *)
(*-------------------------------------------------------------------*)
FUNCTION PathCheck(NW : Network; Event : String) : Boolean;
VAR
  LinkMat : PathMat;
  i, k : integer;
BEGIN
  WITH NW DO
  BEGIN
    PathCheck := FALSE;
    FOR i := 1 TO EdgeNum DO
      IF Event[i]<>'2' THEN GraphMat^[I_index^[i],J_index^[i]] := 0;
    SetNodeList(LinkMat[0], 0);
    LinkMat[0, Source] := 1;
    LinkMat[1] := GraphMat^[Source];
    IF OneWayPath(NW, LinkMat, k) THEN
      PathCheck := TRUE;
  END; (* WITH *)
```

```
END; (* PathCheck *)
(*-------------------------------------------------------------------------*)
PROCEDURE CutPath(VAR NW : Network; SetCut : LinkSet; Count : integer);
VAR
  i,j : integer;
BEGIN
  WITH NW DO
  BEGIN
    alpha^[NodeNum+1] := Count;
    j := 1;
    FOR i := 1 TO EdgeNum DO
      IF i IN SetCut THEN
        BEGIN
          UpEdges[i] := '2';
          PathList^[j] := i;
          Inc(j);
        END;
  END; (* WITH *)
END; (* CutPath *)
(*-------------------------------------------------------------------------*)
FUNCTION CutSetSearch(VAR NW : Network; Event : String) : Boolean;
VAR
  LinkMatF, LinkMatR : PathMat;
  ForCut, RevCut, EventSet : LinkSet;
  i, kf, kr, Rcount, Fcount : integer;
  CheckF, CheckR : Boolean;
BEGIN
  ForCut := [];
  RevCut := [];
  EventSet := [];
  kf := 0;
  WITH NW DO
  BEGIN
    kr := NodeNum + 1;
    CutSetSearch := FALSE;
    FOR i := 1 TO EdgeNum DO
      IF Event[i] = '2' THEN
        EventSet := EventSet + [i];
    SetNodeList(LinkMatF[0], 0);
    FOR i := kf TO kr DO
      BEGIN
        Move(LinkMatF[0], LinkMatF[i], NodeNum*SizeOf(Byte));
        Move(LinkMatF[0], LinkMatR[i], NodeNum*SizeOf(Byte));
      END;
    LinkMatF[kf,source] := 1;
    LinkMatR[kr,sink] := 1;
    CheckF := TRUE;
    CheckR := TRUE;
    REPEAT
      IF CheckF THEN
        BEGIN
          Inc(kf);
          AddSetF(NW,LinkMatF,kf,EventSet,ForCut,CheckF);
          IF (LinkMatF[0,sink] > 0) THEN Exit;
        END;  (* Cutset--forward *)
      IF CheckR THEN
        BEGIN
          Dec(kr);
          AddSetR(NW,LinkMatR,kr,EventSet,RevCut,CheckR);
          IF (LinkMatR[NodeNum+1,source] > 0) THEN Exit;
```

```
        END;  (* Cutset--reverse *)
      FOR i := 1 TO NodeNum DO
        IF (LinkMatF[0,i]>0) AND (LinkMatR[NodeNum+1,i]>0) THEN Exit;
    UNTIL (NOT CheckF) AND (NOT CheckR);
    CutSetSearch := TRUE;
    InitialEvent(UpEdges, '1', EdgeNum);
    SetEdgeList(PathList, 0);
    RCount := 0;
    FCount := 0;
    FOR i := 1 TO EdgeNum DO
      BEGIN
        IF i IN ForCut THEN Inc(FCount);
        IF i IN RevCut THEN Inc(RCount);
      END;
    IF (NOT CheckF) AND (NOT CheckR) THEN
      BEGIN
        IF FCount <= RCount THEN CutPath(NW, ForCut, FCount)
        ELSE CutPath(NW, RevCut, FCount);
      END
    ELSE
      BEGIN
        IF (NOT CheckF) THEN CutPath(NW, ForCut, FCount);
        IF (NOT CheckR) THEN CutPath(NW, RevCut, RCount);
      END;
  END;  (* WITH *)
END; (* CutSetSearch *)
(*-------------------------------------------------------------------*)
PROCEDURE OpenNodeData(VAR NW : Network;
                       VAR ND : NodeData);
(* Set up node data pointers on the heap - initialize the edge matrix *)
VAR
  i, j, k : Integer;
BEGIN
  WITH NW, ND DO
    BEGIN
      New(EdgeMat);
      New(EdgesOn);
      New(EdgesOff);
      New(Gamma);
      MaxDegree := 1;
      SetAdjacencyMat(EdgeMat);
      FOR i := 1 TO NodeNum DO
        BEGIN
          k := 1;
          FOR j := 1 TO NodeNum DO
          IF GraphMat^[j,i] <> 0 THEN
            BEGIN
              EdgeMat^[i,k] := GraphMat^[j,i];
              IF MaxDegree < k THEN
                MaxDegree := k;
              Inc(k);
            END;
        END;
    END; (* WITH *)
END; (* OpenNodeData *)
(*-------------------------------------------------------------------*)
PROCEDURE CloseNodeData(VAR ND : NodeData);
(* Restore node data pointers to the pre-nodedata heap status *)
BEGIN
  WITH ND DO
```

```
    BEGIN
      Dispose(Gamma);
      Dispose(EdgesOff);
      Dispose(EdgesOn);
      Dispose(EdgeMat);
  END; (* WITH *)
END; (* CloseNodeData *)
(*-------------------------------------------------------------------*)
PROCEDURE SetWorkingEdges(    Event : String;
                          VAR ND    : NodeData);
(* Before backfitting, constructs On and Off edgelists for the event *)
VAR
  i : Integer;
BEGIN
  WITH ND DO
    BEGIN
      SetEdgeList(EdgesOn, 0);
      SetEdgeList(EdgesOff, 0);
      FOR i := 1 TO Length(Event) DO
        BEGIN
          IF Event[i] = '2' THEN
            EdgesOn^[i]  := 1
          ELSE IF Event[i] = '0' THEN
            EdgesOff^[i] := 1;
        END;
    END; (* WITH *)
END; (* SetWorkingEdges *)
(*-------------------------------------------------------------------*)
FUNCTION BackFitProb(    Up   : EdgeList;
                         Down : EdgeList;
                     VAR ND   : NodeData;
                         Prob : Real)     : Real;
(* Calculate backfitted probability for the specified node sub-event *)
VAR
  Prob0 : Real;
  i     : Integer;
BEGIN
  Prob0 := Prob;
  WITH ND DO
    BEGIN
      IF (NumOn = 0) AND (NumOff = 0) THEN
        BEGIN
          BackFitProb := 1.0;
          Exit;
        END;
      FOR i := 1 TO MaxDegree DO
        BEGIN
          IF Up[i] = 1 THEN
            Prob0 := Prob0 * Gamma^[i];
          IF Down[i] = 1 THEN
            Prob0 := Prob0 * (1.0 - Gamma^[i]);
        END;
      IF NumOn = 0 THEN
        BackFitProb := 1.0 + Prob0 - Prob
      ELSE
        BackFitProb := Prob0;
    END; (* WITH *)
END; (* BackFitProb *)
(*-------------------------------------------------------------------*)
FUNCTION BackFitNodes(VAR NW    : Network;
```

```
                      VAR ND    : NodeData;
                          Event : String) : Real;
(* Calculate probability of current event backfitted over all nodes *)
VAR
  Prob0, Prob : Real;
  i, j, k     : Integer;
  Up, Down    : EdgeList;
BEGIN
  SetWorkingEdges(Event, ND);
  WITH NW, ND DO
    BEGIN
      Prob := Alpha^[Source];
      SetEdgeVect(Gamma, 0);
      FOR i := 1 TO NodeNum DO
        BEGIN
          Prob0 := Alpha^[i];
          NumOff := 0;
          NumOn := 0;
            FOR j := 1 TO MaxDegree DO
              BEGIN
                Up[j]   := 0;
                Down[j] := 0;
                k := EdgeMat^[i,j];
                IF k > 0 THEN
                  BEGIN
                    Gamma^[j] := Beta^[k];
                    Up[j] := EdgesOn^[k];
                    Down[j] := EdgesOff^[k];
                    Inc(NumOn, Up[j]);
                    Inc(NumOff, Down[j]);
                  END;
              END;
          Prob := Prob * BackFitProb(Up, Down, ND, Prob0);
        END;
      BackFitNodes := Prob;
  END; (* WITH *)
END; (* BackFitNodes *)
(* ------------------------------------------------------------- *)
PROCEDURE AlphaFileError(s : String);
(* Indicate desired format of node probabilities in an AlphaFile *)
BEGIN
  CursorOff;
  QuickPopUp(10, 4, 70, 23, 2, Yellow, Red, '');
  Writeln(Bell);
  Writeln('  ERROR - INCORRECT DATA IN ALPHA FILE');
  Writeln('          ' + s);
  Writeln;
  Writeln('  File should contain node data in ASCII form.');
  Writeln('  First line:  NumNodes (1 integer)');
  Writeln('  Later lines: Specifications for individual edges');
  Writeln('               NodeNumber    Reliability');
  Writeln('  Example:');
  Writeln('                                 ');
  Writeln('                 4                 NOTE:  There are');
  Writeln('                 1  0.925         exactly NumNodes');
  Writeln('                 2  0.853         (4)  later lines');
  Writeln('                 3  0.274         after the header');
  Writeln('                 4  0.992         (the first line)');
  Writeln('                                 ');
  Pause('  Press any key to continue...');
```

```
  CursorOn;
  ClosePopUp;
END; (* AlphaFileError *)
(* ------------------------------------------------------------ *)

PROCEDURE EdgeFileError(s : String);
(* Indicate desired format of the network data in an EdgeFile *)
BEGIN
  CursorOff;
  QuickPopUp(10, 4, 70, 23, 2, Yellow, Red, '');
  Writeln(Bell);
  Writeln('  ERROR - INCORRECT DATA IN EDGE FILE');
  Writeln('          ' + s);
  Writeln;
  Writeln('  File should contain edge data in ASCII form.');
  Writeln('  First line:  Network information (4 integers)');
  Writeln('               Source   Sink   NumNodes   NumEdges');
  Writeln('  Later lines: Specifications for individual edges');
  Writeln('               StartNode    EndNode    Reliability');
  Writeln('  Example:');
  Writeln('                                    ');
  Writeln('                 1   4   4   5       NOTE:  There are');
  Writeln('                 1   2   0.925       exactly NumEdges');
  Writeln('                 1   3   0.853       (5)  later lines');
  Writeln('                 2   3   0.274       after the header');
  Writeln('                 2   4   0.992       (the first line)');
  Writeln('                 3   4   0.806       for this network');
  Writeln('                                    ');
  Pause('  Press any key to continue...');
  ClosePopUp;
  CursorOn;
END; (* EdgeFileError *)
(* ------------------------------------------------------------------ *)
PROCEDURE DataIndexError(s0, s1 : String);
(* Indexing error between otherwise compatible EdgeFile and BetaFile *)
BEGIN
  CursorOff;
  QuickPopUp(10, 4, 70, 12, 2, Yellow, Red, '');
  Writeln(Bell);
  Writeln('  ERROR - INCOMPATABLE DATA IN FILES');
  Writeln('          ' + s0);
  Writeln('           and');
  Writeln('          ' + s1);
  Writeln('  The graph size data is in agreement but');
  Writeln('  the indexing arrangements are different');
  Pause('  Press any key to continue...');
  ClosePopUp;
  CursorOn;
END; (* DataIndexError *)
(* ------------------------------------------------------------ *)
PROCEDURE DataConflictError(VAR NW   : Network;
                                s    : String;
                                m, n : Integer);
(* Incompatible files - differences in the reported network size *)
BEGIN
  CursorOff;
  QuickPopUp(10, 4, 70, 12, 2, Yellow, Red, '');
  Writeln(Bell);
  Writeln('  ERROR - INCOMPATABLE DATA IN FILE');
  Writeln('          ' + s);
```

```
  Writeln;
  WITH NW DO
    BEGIN
      IF m <> NodeNum THEN
        Writeln(m : 12 , ' nodes reported: should be ', NodeNum);
      IF n <> EdgeNum THEN
        Writeln(n : 12 , ' edges reported: should be ', EdgeNum);
    END; (* WITH *)
  Writeln;
  Pause('  Press any key to continue...');
  ClosePopUp;
  CursorOn;
END; (* DataConflictError *)
(* ------------------------------------------------------------- *)
PROCEDURE EventFileError(s : String);
(* Indicate the desired format of the st-path data in an EventFile *)
BEGIN
  CursorOff;
  QuickPopUp(10, 4, 70, 23, 2, Yellow, Red, '');
  Writeln(Bell);
  Writeln('  ERROR - INVALID DATA FORMAT IN EVENT FILE' + Bell);
  Writeln('  FILE: - ', s);
  Writeln('  File should contain edge event data in ASCII form.');
  Writeln('  First line:  Network information (5 integers)');
  Writeln('               Source  Sink  NumNodes  NumEdges  MaxHops');
  Writeln('  Later lines: Success/Failure events');
  Writeln('               SF_Flag  Event_String (length = NumEdges),');
  Writeln('  Example:');
  Writeln('                                    ');
  Writeln('                 1  2  4  12  3  ');
  Writeln('                 1 111112111111  ');
  Writeln('                 2 111110212111      NOTE: Each string');
  Writeln('                       .             length is exactly');
  Writeln('                       .             NumEdges (here 12)');
  Writeln('                                     in this example. ');
  Writeln('               -3 110110210211  ');
  Writeln('                                    ');
  Writeln(Bell);
  Pause('  Press any key to continue...');
  ClosePopUp;
  CursorOn;
END; (* EventFileError *)
(* ----- No initialization ----- *)
END.
```

D.6 IMPLEMENTATION OF THE TCA

D.6.1 **Program TCPTR.PAS**

```
{$X+}
{$S+}
{$M 65520,0,655360}
PROGRAM TCPTR;
(*   ------------------------------------------------------------
   Driver program for Theologou-Carlier network analysis unit
   ------------------------------------------------------------
   Program implementing the Theologou-Carlier algorithm in IEEE
   Transactions on Reliability, Vol 40 (June, 1991), pp 210-217.
```

```
   This is a test of the TCUPTR.PAS unit, including the data
   structures for jamming and hidden links.  From jamming data
   the graph structure is calculated with its link reliabilities
   (Betas).  This is shown for any user-supplied source-sink
   selection, with an option to estimate the st-reliability for
   this selected source-sink pair. This portion requests a value
   of the node reliability (Alpha, a constant), and calculates
   the exact value of the st-reliability.  Stack overflow is
   avoided by using pointer variables.
   ------------------------------------------------------------
   Note that in the unit it is not required that the node failure
   Alpha be a constant.  Alpha can very from node to node, although
   in this test program the node failure rate is a constant.
   ------------------------------------------------------------
*)
USES
  Crt, Dos, TCUptr, PopMenus, Dir_Menu,
  Strings, Keyboard, FileChck;
VAR
  g : Graph;
  NumJams : Integer;
  Hidden : EdgeSet;
  SINR : TriplePointer;
  Threshold, SigmaL : Real;
(* ---------------------------------------------------------------------
*)
PROCEDURE BuildGraph(VAR g : Graph; VAR h : EdgeSet;
                     kj : Integer; Thresh, Sigma : Real);
{ Initialize the graph g from SNRs supplied in array SINR.  This
  procedure also does the hidden edges for the graph.}
VAR
   Temp, arg : Real;
   k, n, nv, sv, tv : Integer;
BEGIN  {BuildGraph}
  WITH g DO
  BEGIN
    n  := 0;
    h.n := 0;
    Vert := [];
    nv := NumNodes;
    FOR sv := 1 TO nv DO      {loop on source vertex}
      FOR tv := 1 TO nv DO    {loop on terminal vertex}
        IF sv <> tv THEN
          BEGIN  {create an edge}
            arg := (SINR^[sv, tv, kj] - Thresh) / Sigma;
            Temp := Pfunction(arg);
            IF arg >= -3.0 THEN
              BEGIN  {there is a viable edge}
                IF arg >= 0.0 THEN
                  BEGIN {unhidden edge}
                    Inc(n);
                    e[n].Start := sv;
                    e[n].Stop := tv;
                    e[n].Beta := Temp;
                    IF e[n].Start > NumNodes THEN
                      NumNodes := e[n].Start;
                    IF e[n].Stop > NumNodes THEN
                      NumNodes := e[n].Stop;
                    Vert := Vert + [e[n].Start, e[n].Stop];
                    nb[e[n].Start] := nb[e[n].Start] + [e[n].Stop];
```

```
                END;  {Unhidden edge}
              IF arg < 0.0 THEN
                BEGIN {hidden edge}
                  Inc(h.n);
                  h.e[h.n].Start := sv;
                  h.e[h.n].Stop := tv;
                  h.e[h.n].Beta := Temp;
                END;  {Hidden edge}
            END; {there is a viable edges}
        END;  {create an edge}
    NumEdges := n;
  END; (* WITH *)
END;  {BuildGraph}
(* ---------------------------------------------------------------- *)
PROCEDURE ReadTripleData(TripPtr : TriplePointer; VAR g : Graph);
LABEL
  READ_ERROR;
VAR
  SNRFile : Text;
  i, j, k : Integer;
  Msg, s, FileSpec, FileStr, ExtStr, SNRStr, DirSpec : String;
BEGIN
  FileSpec := '*.SNR';
  DirSpec := '';
  Msg := '  << SNRFile Selection (F1 for HELP)';
  SNRStr := DirectoryMenu(DirSpec, FileSpec, Msg);
  MakeStrUpper(SNRStr);
  {$V-} Fsplit(SNRStr, DirSpec, FileStr, ExtStr); {$V+}
  IF NOT CheckOldFile(SNRFile, SNRStr) THEN
    GoTo READ_ERROR;
  {$I-} Readln(SNRFile, s); {$I+}
  IF (IOResult <> 0) OR (ExitCode <> 0) THEN
    GoTo READ_ERROR;
  {$I-} Readln(SNRFile, i, g.NumNodes, NumJams); {$I+}
  WHILE NOT EOF(SNRFile) DO
    BEGIN
      {$I-} Read(SNRFile, i, j); {$I+}
      IF NumJams = 1 THEN
        {$I-} Readln(SNRFile, TripPtr^[i,j,1]) {$I+}
      ELSE
        BEGIN
          FOR k := 1 to NumJams - 1 DO
            {$I-} Read(SNRFile, TripPtr^[i,j,k]); {$I+}
          {$I-} Readln(SNRFile, TripPtr^[i,j,NumJams]); {$I+}
        END;
    END; (* WHILE *)
  Close(SNRFile);
  Exit;
READ_ERROR:
  Close(SNRFile);
END; (* ReadTripleData *)
(* ---------------------------------------------------------------- *)
PROCEDURE GetSignalData(VAR Threshold, SigmaL : Real);
VAR
  Ch : Char;
  CursorX, CursorY : Byte;
BEGIN
  QuickPopUp(10, 5, 70, 20, 2, White, Blue, '');
  Writeln;
  Writeln('  PARAMETERS:');
```

```
  Writeln;
  Writeln('    SNR Threshold value in dB:     ', Threshold : 10 : 4);
  Writeln('    Propagation sigma (S+J) in dB: ', SigmaL : 10 : 4);
  Writeln;
  CursorX := WhereX;
  CursorY := WhereY;
  Write('  Accept these parameters (Y/N) ? ');
  Ch :=  UpCase(GetKey);
    IF Ch = 'Y' THEN
     BEGIN
        ClosePopUp;
        Exit;
     END;
  GoToXY(CursorX, CursorY);
  Writeln('  Enter new data (just Enter to leave an item unchanged)');
  Writeln;
  GetRealNo( '    Enter threshold value (dB):________ ', Threshold);
  GetRealNo( '    Enter propagation sigma (dB):______ ', SigmaL);
  ClosePopUp;
END; (* GetSignalData *)
(* ---------------------------------------------------------------- *)
PROCEDURE TestSTRel(VAR g : Graph; VAR h: EdgeSet);
VAR
  k, JamID, temp : Integer;
  arg, t, Alpha0 : Real;
  s1, s2, Again, More : String;
BEGIN
  Again := 'Y';
  Str(g.NumNodes, s1);
  Str(NumJams, s2);
  QuickPopUp(10, 4, 70, 20, 2, White, Blue, '');
  WHILE Again <> 'N' DO
    BEGIN
      ClrScr;
      Writeln;
      Writeln('  Enter source and sink nodes');
      Writeln;
      GetPosInt('    Source node (1-' + s1 + ')_____ ', temp);
      if temp > g.NumNodes then g.source := 1 else g.source := temp;
      GetPosInt('    Sink node (1-' + s1 + ')_______ ', temp);
        if temp > g.NumNodes then g.Sink := g.numNodes else g.Sink :=
temp;
      GetPosInt('    Jammer case (1-' + s2 + ')______ ', JamID);
      Writeln;
      arg := SINR^[g.Source, g.Sink, JamID];
      t := BetaQ(arg, Threshold, SigmaL);
      Writeln('    Link reliability (Beta): ', t : 12 : 8);
      Writeln;
      Write('  Calculate the st-reliability (Y/N) ? ');
      More := UpCase(GetKey);
      Writeln(More);
      IF More = 'Y' THEN
        BEGIN
          Writeln;
          GetRealNo('    Node reliability (Alpha) << ?  ', Alpha0);
          CursorOff;
          FOR k := 1 TO NODEMAX DO
            BEGIN
              g.Alpha[k] := Alpha0;
              g.nb[k] := [];
```

```
            END;
          BuildGraph(g, h, JamID, Threshold, SigmaL);
          Writeln('    ST-reliability: ', TCSTReliabilU(g, h) : 12 : 8);
        END;
      CursorOn;
      Writeln;
      Write('  Another node pair (Y/N) ? ');
      Again := UpCase(GetKey);
    END; (* WHILE *)
  ClosePopUp;
END; (* TestSTRel *)
(* ------------------------------------------------------------- *)
PROCEDURE initializeSNRData( var RHO : triplePointer);
(* Initialize SNR data to large negative number *)
const
     bigNeg = -99.9;
var
   i, j, k : integer;
begin
     for i := 1 to NodeMax do
         for j := 1 to nodeMax do
             for k := 1 to maxJam do
                 RHO^[i, j, k] := bigNeg;
end;

(* ---------------------- Main Program ----------------------------- *)
BEGIN
  ClrScr;
  New(SINR);
  initializeSNRData( SINR);
  Threshold := 0.0;
  SigmaL := 10.0;
  ReadTripleData(SINR, g);
  GetSignalData(Threshold, SigmaL);
  TestSTRel(g, Hidden);
  Dispose(SINR);
END. (* Main Program *)
```

D.6.2 Unit TCUPTR.PAS

```
UNIT TCUPTR;
(*   ------------------------------------------------------------------
-
   Unit for Theologu-Carlier network analysis - node failures OK
   ------------------------------------------------------------------
   Graph description using pointers to save stack space
   ------------------------------------------------------------------
   Exact calculation
   ------------------------------------------------------------------
   Unit for graph reduction functions implementing the Theologou-Carlier
   algorithm in IEEE Transactions on Reliability, Vol 40 (June, 1991),
   pp 210-217.  Includes data structures for jamming and hidden links
    ------------------------------------------------------------------
*)
INTERFACE
CONST
  MAXJAM = 10;           {Maximum number of jammers}
  NODEMAX = 15;          {Maximum number of nodes}
```

```
  EDGEMAX = 225;         {Maximum number of edges}
TYPE
  DegreeType = ARRAY[1..NODEMAX] OF Integer;  {List of vertex degrees}
  GraphSet = SET OF 1..NODEMAX;               {Set of vertices}
  Edge = RECORD                               {Edge in a graph}
      Start,                                  {Start vertex}
      Stop : 1..NODEMAX;                      {Stop vertex }
      Beta : Real                             {Edge reliabilities}
    END; { Edge }
  EdgeSet = RECORD
      e : ARRAY[1..EDGEMAX] OF Edge;
      n : Integer;
    END; (* EdgeSet *)
  pGraph = ^Graph;
  Graph = RECORD                            {Describes a graph}
      Vert : GraphSet;                      {Set of graph vertices}
      Source,                               {Source vertex}
      Sink : Integer;                       {Sink vertex}
      InDegree,                             {In degree of each vertex}
      OutDegree : DegreeType;               {Out degree of each vertex}
      nb : ARRAY[1..NODEMAX] OF GraphSet; {Edge(i,j) puts j in nb[i]}
      NumEdges : Integer;                   {Number of edges in the graph}
      NumNodes : Integer;         {Largest numbered vertex in the graph}
      e : ARRAY[1..EDGEMAX] OF Edge;  {Describes all edges in the graph}
      Alpha : ARRAY[1..NODEMAX] OF Real;  {Node reliabilities}
    END; { Graph }
   Vectj = ARRAY[1..MAXJAM] OF Real;
   Matrxi = ARRAY[1..NODEMAX] OF Vectj;
   TripleArray = ARRAY[1..NODEMAX] OF Matrxi;
   TriplePointer = ^TripleArray;
FUNCTION   Pfunction(z : Real) : Real;
FUNCTION   BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
FUNCTION   Connected( VAR g : Graph) : Boolean;
FUNCTION   TCSTReliabilU(VAR g : Graph; VAR h: EdgeSet) : Real;
(* ------------------------------------------------------------------ *)
IMPLEMENTATION
(*   ------------------------------------------------------------------
   Most of the graph processing functions below are modifications of the
  earlier Page-Perry procedures for network reduction and factorization,
 but rewritten according to Theologu-Carlier in order to account for the
   possibility of node failures.  Data structures for jamming and hidden
   links are used in the calculation.
   ------------------------------------------------------------------*)
USES
  Crt, Dos, KeyBoard, TextScrn, Strings, PopMenus, FileChck;
(* ------------------------------------------------------------------ *)
FUNCTION Pfunction(z : Real) : Real;
```

[For a listing of this function, see Section D.1.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
VAR
   Temp : Real;
BEGIN
   Temp := (RhodB - MargindB) / SigmadB;
   BetaQ := Pfunction(Temp);
END; (* BetaQ *)
(*------------------------------------------------------------------*)
PROCEDURE FindDegree (VAR g : Graph);
```

```
{ Determine the degree of each vertex in the graph }
VAR
  i : Integer;  {Edge number}
BEGIN  {FindDegree}
  WITH g DO
    BEGIN
      FOR i := 1 TO NumNodes DO
        BEGIN
          InDegree[i] := 0;
          OutDegree[i] := 0;
        END;
      FOR i := 1 TO NumEdges DO
        BEGIN
          Inc(OutDegree[e[i].Start]);
          Inc(InDegree[e[i].Stop]);
        END;
    END; (* WITH *)
END; {FindDegree}
(*-------------------------------------------------------------------------*)
PROCEDURE Delete (VAR g : Graph; n : Integer);
{ Deletes edge n from the graph g.  Degrees and neighbors are changed.}
VAR
  j: Integer;
  u, v : Integer;  {Endpoints of the deleted edge}
BEGIN  {Delete}
  WITH g DO
    BEGIN
      u := e[n].Start;
      v := e[n].Stop;
      nb[u] := nb[u] - [v];
      Dec(InDegree[v]);
      Dec(OutDegree[u]);
      FOR j := n TO NumEdges - 1 DO
        e[j] := e[j + 1];
      Dec(NumEdges);
    END; (* WITH *)
END;  {Delete}
(*-------------------------------------------------------------------------*)
PROCEDURE CleanSink (VAR g : Graph);
{ Remove all edges in g that have the sink as starting vertex.}
VAR
  j : Integer;
BEGIN   {CleanSink}
  WITH g DO
    BEGIN
      FOR j := NumEdges DOWNTO 1 DO
        IF e[j].Start = Sink THEN
          Delete(g, j);
    END; (* WITH *)
END;   {CleanSink}
(*-------------------------------------------------------------------------*)
PROCEDURE CleanSource (VAR g : Graph);
{ Remove all edges in g that have the source as terminating vertex.}
VAR
  j : Integer;
BEGIN  {CleanSource}
  WITH g DO
    BEGIN
      FOR j := NumEdges DOWNTO 1 DO
        IF e[j].Stop = Source THEN
```

```
          Delete(g, j);
    END; (* WITH *)
END;  {CleanSource}
(*-------------------------------------------------------------------*)
PROCEDURE CleanUp (VAR g : Graph);
{ Eliminates all dead end and false start vertices in g.}
VAR
  j, u : Integer;
   Reduced : Boolean;  {Set false if a dead end or false start vertex
found}
BEGIN   {CleanUp}
  CleanSource(g);
  CleanSink(g);
  WITH g DO
    BEGIN
      REPEAT
        Reduced := TRUE;
        FOR u := 1 TO NumNodes DO
          IF (u <> Source) AND (u <> Sink) THEN
            IF (InDegree[u] = 0) or (OutDegree[u] = 0) THEN
              IF (u IN Vert) THEN
                BEGIN {eliminate vertex u}
                  Reduced := FALSE;
                  FOR j := NumEdges DOWNTO 1 DO
                    IF (e[j].Start = u) or (e[j].Stop = u) THEN
                      Delete(g, j);
                  Vert := Vert - [u]
                END;  {eliminate vertex u}
      UNTIL Reduced
    END; (* WITH *)
END;  {CleanUp}
(*-------------------------------------------------------------------*)
PROCEDURE ForwardSimplify (VAR g : Graph; VAR Simplified : Boolean);
{ If one exists, eliminates a nonnecessary edge coming
  into a vertex and sets Simplified to TRUE.}
VAR
  j : Integer;
  v : Integer;  {Initial vertex for an edge}
  w : Integer;  {Terminal vertex of edge out of v}
BEGIN {ForwardSimplify}
  WITH g DO
    BEGIN
      FOR v := 1 TO NumNodes DO
        IF (OutDegree[v] = 1) THEN
          BEGIN  {Look for edge antiparallel to the edge out of v.}
            FOR j := 1 TO NumEdges DO
              IF (e[j].Start = v) THEN
                w := e[j].Stop;
            FOR j := NumEdges DOWNTO 1 DO
              IF (e[j].Stop = v) AND (e[j].Start = w) THEN
                BEGIN  {Delete the antiparallel edge.}
                  Delete(g, j);
                  Simplified := TRUE
                END  {Delete the antiparallel edge.}
          END  {Look for edge antiparallel to the edge out of v.}
    END; (* WITH *)
END;  {ForwardSimplify}
(*-------------------------------------------------------------------*)
PROCEDURE BackSimplify (VAR g : Graph; VAR Simplified : Boolean);
{ If one exists, eliminates a nonnecessary edge coming
```

```
  out of a vertex and sets Simplified to TRUE.}
VAR
  j : Integer;
  v : Integer;  {Terminal vertex for an edge}
  w : Integer;  {Initial vertex of edge out of v}
BEGIN  {BackSimplify};
  WITH g DO
    BEGIN
      FOR v := 1 TO NumNodes DO
        IF InDegree[v] = 1 THEN
          BEGIN  {Look for edge antiparallel to the edge into v.}
            FOR j := 1 TO NumEdges DO
              IF (e[j].Stop = v) THEN
                w := e[j].Start;
            FOR j := NumEdges DOWNTO 1 DO
              IF (e[j].Start = v) AND (e[j].Stop = w) THEN
                BEGIN  {Delete the antiparallel edge.}
                  Delete(g, j);
                  Simplified := TRUE
                END  {Delete the antiparallel edge.}
          END  {Look for edge antiparallel to the edge into v.}
    END; (* WITH *)
END;  {BackSimplify}
(*-------------------------------------------------------------------------*)
PROCEDURE SourceSinkRed(VAR g:Graph; VAR Found:Boolean; VAR Factor :
Real);
{ If the sink of graph g has in-degree 1, then it is merged into its
  neighbor and the resulting sink is cleaned of out-edges. If the souce
has
  out-degree 1, then the parallel result occurs.  Factor is returned as
the
  appropriate multiplying factor for the graph.}
VAR
  j : Integer;            {Possible edge incident to source or sink}
  OldSink :   Integer;    {Original sink vertex}
  OldSource : Integer;    {Original source vertex}
  IntoSink :    Integer;  {Edge Into the sink}
  OutOfSource : Integer;  {Edge out of the source}
BEGIN  {SourceSinkRed}
  WITH g DO
    BEGIN
      Found := FALSE;
      IF InDegree[Sink] = 1 THEN
        BEGIN  {Merge the sink Into its adjacent vertex.}
          Found := TRUE;
          FOR j := 1 TO NumEdges DO
            IF e[j].Stop = Sink THEN
              IntoSink := j;
          OldSink := Sink;
          Sink := e[IntoSink].Start;
          Factor := Factor * e[IntoSink].Beta * Alpha[Sink];
          Alpha[sink] := 1.0;
          Delete(g,IntoSink);
          Vert := Vert - [OldSink];
          CleanSink(g);
        END;  {Merge the sink into its adjacent vertex.}
      IF (OutDegree[Source] = 1) AND (Source <> Sink) THEN
        BEGIN  {Merge the source Into its adjacent vertex.}
          Found := TRUE;
          FOR j := 1 TO NumEdges DO
```

```
            IF e[j].Start = Source THEN
              OutOfSource := j;
          OldSource := Source;
          Source := e[OutOfSource].Stop;
          Factor := e[OutOfSource].Beta * Factor * Alpha[Source];
          Alpha[Source] := 1.0;
          Delete(g, OutOfSource);
          Vert := Vert - [OldSource];
          CleanSource(g);
        END;  {Merge the source into its adjacent vertex.}
    END; (* WITH *)
END; {SourceSinkRed}
(*-------------------------------------------------------------------*)
PROCEDURE InOutDeg1Red (VAR g : Graph; VAR Found : Boolean);
{ G is scanned to find a vertex with in-degree and out-degree 1. If such
a
  vertex is found, it it removed and the resulting graph is simplified.}
VAR
  j : Integer;          {Graph edge}
  u : Integer;          {Graph vertex (with possible in/out degree 1)}
  InRel : Real;         {Reliability of edge into u}
  OutRel : Real;        {Reliability of edge out of u}
  DoubleRel : Real;     {Reliability of both edges in sequence}
  InitV : Integer;      {Initial vertex of edge into u}
  TermV : Integer;      {Terminal vertex of edge out of u}
BEGIN {InOutDeg1Red}
  WITH g DO
    BEGIN
      FOR u := 1 TO NumNodes DO
        IF (InDegree[u] = 1) AND (OutDegree[u] = 1) THEN
          BEGIN  {Vertex u has in- and out-degree 1.  Eliminate it.}
            Found := TRUE;
            FOR j := NumEdges DOWNTO 1 DO
              IF e[j].Stop = u THEN
                BEGIN  {This is the edge into u.}
                  InitV := e[j].Start;
                  InRel := e[j].Beta;
                  Delete(g, j)
                END;  {This is the edge into u.}
            FOR j := NumEdges DOWNTO 1 DO
              IF e[j].Start = u THEN
                BEGIN  {This is the edge out of u.}
                  TermV := e[j].Stop;
                  OutRel := e[j].Beta;
                  Delete(g, j);
                END;  {This is the edge out of u.}
            DoubleRel := InRel * OutRel * Alpha[u];
            Vert := Vert - [u];
            IF TermV <> InitV THEN
              IF TermV IN nb[InitV] THEN
                BEGIN {Redo reliability of edge from InitV to TermV.}
                  FOR j := 1 TO NumEdges DO
                    IF (e[j].Start = InitV) AND (e[j].Stop = TermV) THEN
                            e[j].Beta := e[j].Beta * (1 - DoubleRel) +
DoubleRel;
                END  {Redo reliability of edge from InitV to TermV.}
              ELSE
                BEGIN  {Construct a new edge from InitV to TermV}
                  NumEdges := NumEdges + 1;
                  e[NumEdges].Start := InitV;
```

```
                  e[NumEdges].Stop := TermV;
                  e[NumEdges].Beta := DoubleRel;
                  nb[InitV] := nb[InitV] + [TermV];
                  Inc(InDegree[TermV]);
                  Inc(OutDegree[InitV]);
                END;  {Construct a new edge from InitV to TermV}
            Exit;
          END; {Vertex u has in- and out-degree 1.  Eliminate it.}
    END; (* WITH *)
END; {DegTwoRed}
(*-------------------------------------------------------------------------*)
PROCEDURE Contract (VAR g : Graph; NewSink : Integer);
{ Contracts the sink of graph g into the vertex NewSink.}
VAR
  v : Integer;        {Graph vertex}
  j : Integer;        {Graph edge}
  InSink : Integer;   {Edge from v into the old sink}
  InNewSink : Integer;{Edge from v into the new sink}
  Parallel : Boolean; {True if an edge go from v to the NewSink}
BEGIN {Contract}
  WITH g DO
    BEGIN
      FOR v := 1 TO NumNodes DO
        IF (Sink IN nb[v]) THEN
          BEGIN  {There is an edge from v to the sink.  Change it.}
            Parallel := FALSE;
            FOR j := 1 TO NumEdges DO
              IF (e[j].Start = v) AND (e[j].Stop = Sink) THEN
                InSink := j
              ELSE IF (e[j].Start = v) AND (e[j].Stop = NewSink) THEN
                BEGIN  {There is also an edge from v to the NewSink.}
                  Parallel := TRUE;
                  InNewSink := j;
                END;  {There is also an edge from v to the NewSink.}
              IF Parallel THEN
               BEGIN {Eliminate edge (v,sink). Change reliability of
                    (v,NewSink)}
                  e[InNewSink].Beta := e[InNewSink].Beta
                         * (1 - e[InSink].Beta)  + e[InSink].Beta;
                  Delete(g, InSink)
                END  {Eliminate edge InSink.
                         Change reliability of InNewSink.}
              ELSE
                BEGIN {Change edge InSink to have terminal vertex NewSink.}
                e[InSink].Stop := NewSink;
                Inc(InDegree[NewSink]);
                Dec(InDegree[Sink]);
               END;
             nb[v] := (nb[v] + [NewSink]) - [Sink];
          END;
      Vert := Vert - [Sink];
      Sink := NewSink;
      CleanSink(g);
    END; (* WITH *)
END;  {Contract}
(* ------------------------------------------------------------------- *)
FUNCTION findEdge( var g : Graph; i, j : integer) : integer;
VAR k, l : integer;
BEGIN
     WITH g DO
     FOR k := 1 TO numEdges DO
```

```
          IF (e[k].start = i) AND (e[k].stop = j) THEN
             l := k;
     findEdge := l;
END;
(*-------------------------------------------------------------------------*)
FUNCTION Connected ( VAR g : Graph) : Boolean;
{ Determine if the sink can be reached from the source }
{ Combined with sinkEdge to find the edge on the shortest path }
VAR
  comp : GraphSet;   {Vertices so far reachable from the source}
  u : Integer;       {Possible vertex in comp}
  OldSet : GraphSet; {Comp on the last pass through the graph}
  Changed : Boolean; {True when a new vertex is added to comp}
BEGIN {BFS}
  WITH g DO
    BEGIN
      comp := [Source];
      REPEAT
            OldSet := comp;
            Changed := FALSE;
            FOR u := 1 TO NumNodes DO
                if u in comp then
                   comp := comp + nb[u];
            IF comp <> OldSet THEN
               Changed := TRUE;
      UNTIL (NOT Changed);
      Connected := Sink IN comp
    END; (* WITH *)
END;  {BFS}
(*-------------------------------------------------------------------------*)
PROCEDURE  SinkEdge(VAR  g  :  Graph;  VAR  k  :  Integer;  VAR  InitVert  :
Integer);
{ Find an edge k into the sink. Initial vertex is InitVert.}
VAR
  j : Integer;     {Edge number}
BEGIN   {SinkEdge}
  WITH g DO
    BEGIN
      FOR j := 1 TO NumEdges DO
        IF (e[j].Stop = Sink) THEN
          BEGIN
            k := j;
            InitVert := e[j].Start
          END;
    END; (* WITH *)
END;  {SinkEdge}
(* ------------------------------------------------------------------- *)

FUNCTION UpperProb(pg : pGraph) : Real;
{ Returns the reliability of the graph g.}
VAR
  Reducible : Boolean;   {True if the graph was just reduced}
  p : Real;              {Factor for probability of the reduced graph}
  MarkedEdge : Integer;  {Edge used for factoring}
  ProbEdge : Real;       {Probability of edge used for factoring}
  InitVert : Integer;    {Endpoint of factored edge}
  p1 : Real;             {Probability of g with edge removed}
  pLink : Real;
  alphaInit : Real;
  pLocalGraph : pGraph;
BEGIN  {UpperProb}
     pLocalGraph := new( pGraph);
     pLocalGraph^ := pg^;        { Copy the graph into local variable }
```

```
  WITH pLocalGraph^ DO
    BEGIN
      p := 1.0;
      REPEAT
        Reducible := FALSE;
        CleanUp( pLocalGraph^ );
        IF (Source <> Sink) AND (InDegree[Sink] > 0) AND
                                  (OutDegree[Source] > 0) THEN
          BEGIN
            SourceSinkRed( pLocalGraph^, Reducible, p);
            IF NOT Reducible THEN
              BEGIN  {No source or sink reduction was possible}
                BackSimplify( pLocalGraph^, Reducible);
                ForwardSimplify( pLocalGraph^, Reducible);
                InOutDeg1Red( pLocalGraph^, Reducible);
              END
          END
      UNTIL NOT Reducible;
      p := p * Alpha[Sink] * alpha[source];
      Alpha[Sink] := 1.0;
      alpha[source] := 1.0;
      IF (Source = Sink) THEN
        UpperProb := p
      ELSE IF (InDegree[Sink] = 0) OR (OutDegree[Source] = 0) THEN
        UpperProb := 0
      ELSE
        BEGIN { Factor the graph -- no more reductions are possible}
          SinkEdge( pLocalGraph^, MarkedEdge, InitVert);
          alphaInit := alpha[initVert];
          ProbEdge := e[MarkedEdge].Beta;
          pLink := ProbEdge * AlphaInit;    {  Alpha of terminal is 1 }
          Delete( pLocalGraph^, MarkedEdge);
          IF NOT Connected( pLocalGraph^) THEN
             p1 := 0
          ELSE BEGIN
              {alpha of sink is 1, after deletion of the edge, alpha at
the
                   other end of the deleted edge is calculated as the
following}
            alpha[initVert] := alphaInit * (1 - probEdge)
                                 / (1 - probEdge * alphaInit);
               { alpha of sink still 1 }
              p1 := UpperProb( pLocalGraph);
            END;
          contract( pLocalGraph^, initVert);
          alpha[sink] := 1.0;
          upperProb := p * ( pLink * upperProb( pLocalGraph) +
                               (1 - pLink) * p1 );
        END  {Factor the graph -- no more reductions are possible}
    END; (* WITH *)
    dispose( pLocalGraph);
END; {UpperProb}
(* ------------------------------------------------------------------ *)
FUNCTION HiddenProb(VAR g : Graph; VAR h: EdgeSet) : Real;
VAR
  i : Integer;
  t : Real;
BEGIN
  WITH g DO
    BEGIN
```

```
      t := 0.0;
      FOR i := 1 TO h.n DO
        IF (h.e[i].Start = Source) AND (h.e[i].Stop = Sink) THEN
          t := h.e[i].Beta;
      HiddenProb := t;
    END; (* WITH *)
END; (* HiddenProb *)
(* ------------------------------------------------------------------ *)
FUNCTION TCSTReliabilU(VAR g : Graph; VAR h: EdgeSet) : Real;
VAR
  pa : Real;
BEGIN
  WITH g DO
    BEGIN
      FindDegree(g);
      pa := HiddenProb(g, h) * Alpha[Source] * Alpha[Sink];
      TCSTReliabilU := pa + (1 - pa) * UpperProb( @g);
    END; (* WITH *)
END; (* TCSTReliabilU *)
(*---------------- No Initialization ---------------------*)
END.
```

D.7 IMPLEMENTATION OF THE TCA WITH BOUNDS

D.7.1 **Program TCPTRBND.PAS**

```
{$X+}
{$S+}
{$M 65520,0,655360}
PROGRAM TCPTRBND;
(*   ------------------------------------------------------------
   Driver program for Theologou-Carlier network analysis unit
   ------------------------------------------------------------
   Program implementing the Theologou-Carlier algorithm in IEEE
   Transactions on Reliability, Vol 40 (June, 1991), pp 210-217.
   This is a test of the TCUPTRUL.PAS unit, including the data
   structures for jamming and hidden links.  From jamming data
   the graph structure is calculated with its link reliabilities
   (Betas).  This is shown for any user-supplied source-sink
   selection, with an option to estimate the st-reliability for
   this selected source-sink pair. This portion requests a value
   of the node reliability (Alpha, a constant), and attempts to
   calculate upper and lower bounds on the st-reliability.
   ------------------------------------------------------------
   Note that in the unit it is not required that the node failure
   Alpha be a constant.  Alpha can very from node to node, although
   in this test program the node failure rate is a constant.
   ------------------------------------------------------------
   Graph descriptions using pointer arrays to save stack memory. *)
USES
  Crt, Dos, TCptrUL, PopMenus, Dir_Menu,
  Strings, Keyboard, FileChck, timer, commGraf;
VAR
  g : Graph;
  NumJams : Integer;
  Hidden : EdgeSet;
  Threshold, SigmaL : Real;
(* ------------------------------------------------------------------ *)
```

```
PROCEDURE BuildGraph(VAR g : Graph; VAR h : EdgeSet;
                     kj : Integer; Thresh, Sigma : Real);
{ Initialize the graph g from SNRs supplied in unit commGraf.  This
  procedure also does the hidden edges for the graph.
  This version also calculates the link statistics  -- pAverage, pMax }
VAR
   Temp, arg : Real;
   k, n, nv, sv, tv : Integer;
BEGIN  {BuildGraph}
  WITH g DO
  BEGIN
       pAverage := 0.0;
       pMax := 0.0;
    n  := 0;
    h.n := 0;
    Vert := [];
    FOR sv := 1 TO g.numNodes DO      {loop on source vertex}
         FOR tv := 1 TO g.numNodes DO    {loop on terminal vertex}
         IF sv <> tv THEN
         BEGIN  {create an edge}
              arg := (getSNRvalue(sv, tv, kj) - Thresh) / Sigma;
              Temp := Pfunction(arg);
              IF arg >= -3.0 THEN
              BEGIN  {there is a viable edge}
                   IF arg >= 0.0 THEN
                   BEGIN {unhidden edge}
                        Inc(n);
                        e[n].Start := sv;
                        e[n].Stop := tv;
                        e[n].Beta := Temp;
                        Vert := Vert + [e[n].Start, e[n].Stop];
                        nb[sv] := nb[sv] + [tv];
                        if temp > pMax then
                           pMax := temp;
                        pAverage := pAverage + temp;
                   END;  {Unhidden edge}
                   IF arg < 0.0 THEN
                   BEGIN {hidden edge}
                        Inc(h.n);
                        h.e[h.n].Start := sv;
                        h.e[h.n].Stop := tv;
                        h.e[h.n].Beta := Temp;
                   END;  {Hidden edge}
              END; {there is a viable edges}
          END;  {create an edge}
    NumEdges := n;
    pAverage := pAverage / n;
  END; (* WITH *)
END;  {BuildGraph}
(* ---------------------------------------------------------------- *)
PROCEDURE ReadData;
VAR
  SNRFile : Text;
  Msg, s, FileSpec, FileStr, ExtStr, SNRStr, DirSpec : String;
BEGIN
  FileSpec := '*.SNR';
  DirSpec := '';
  Msg := '  << SNRFile Selection (F1 for HELP)';
  SNRStr := DirectoryMenu(DirSpec, FileSpec, Msg);
  MakeStrUpper(SNRStr);
```

```
  {$V-} Fsplit(SNRStr, DirSpec, FileStr, ExtStr); {$V+}
  IF CheckOldFile(SNRFile, SNRStr) THEN
     readSNRfile( SNRFile);
  Close(SNRFile);
  g.numNodes := getNumNodes;
  numJams := getNumJams;
END; (* ReadTripleData *)
(* ------------------------------------------------------------------ *)
PROCEDURE GetSignalData(VAR Threshold, SigmaL : Real);
```

[For a listing of this procedure, see Section D.6.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE TestSTRel(VAR g : Graph; VAR h: EdgeSet);
const
     rFileName : pathStr = 'c:\tp\test\result.txt';
VAR
  k, JamID, temp : Integer;
  arg, t, Alpha0 : Real;
  s1, s2, Again, More : String;
  i : integer;
  resultFile : text;
  tu, tl : real;
BEGIN
  Again := 'Y';
  Str(g.NumNodes, s1);
  Str(NumJams, s2);
  QuickPopUp(10, 4, 70, 20, 2, White, Blue, '');
  WHILE Again <> 'N' DO
    BEGIN
      ClrScr;
      Writeln;
      Writeln('  Enter source and sink nodes');
      Writeln;
      GetPosInt('    Source node (1-' + s1 + ')_____ ', temp);
      if temp > g.NumNodes then g.source := 1 else g.source := temp;
      GetPosInt('    Sink node (1-' + s1 + ')_______ ', temp);
        if temp > g.NumNodes then g.Sink := g.numNodes else g.Sink :=
temp;
      GetPosInt('    Jammer case (1-' + s2 + ')______ ', JamID);
      Writeln;
      arg := getSNRvalue(g.Source, g.Sink, JamID);
      t := BetaQ(arg, Threshold, SigmaL);
      Writeln('    Link reliability (Beta): ', t : 12 : 8);
      Writeln;
      Write('  Calculate the st-reliability (Y/N) ? ');
      More := UpCase(GetKey);
      Writeln(More);
      IF More = 'Y' THEN
        BEGIN
           assign( resultFile, rFileName);
           rewrite( resultFile);
          Writeln;
          GetRealNo('    Node reliability (Alpha) << ?  ', Alpha0);
          CursorOff;
           for i := 0 to 8 do
           begin
                threshold := i;
          FOR k := 1 TO g.numNodes DO
            BEGIN
```

```
              g.Alpha[k] := Alpha0;
              g.nb[k] := [];
            END;
          BuildGraph(g, h, JamID, Threshold, SigmaL);
             writeln( ' Calculating for threshold = ', threshold:3:1);
          startTimer;
          TCSTReliabil(g, h, tU, tL);
          stopTimer;
          Writeln( resultFile, ' Threshold: ', threshold : 4 : 1,
                          '  Upper Bound: ',  tu: 10 : 8);
          writeln( resultFile, ' Time:  ', getElapSeconds:8:3,
                          '  Lower Bound: ',  tL: 10 : 8);
          Writeln( resultFile);
           end;
           close( resultFile);
        END;
      CursorOn;
      Writeln;
      Write('  Another node pair (Y/N) ? ');
      Again := UpCase(GetKey);
    END; (* WHILE *)
  ClosePopUp;
END; (* TestSTRel *)
(* ---------------------- Main Program ----------------------------- *)
BEGIN
  ClrScr;
  Threshold := 0.0;
  SigmaL := 10.0;
  ReadData;
  GetSignalData(Threshold, SigmaL);
  TestSTRel(g, Hidden);
END. (* Main Program *)
```

D.7.2 Unit TCUPTRUL.PAS

```
UNIT TCUPTRUL;
(*    ---------------------------------------------------------------------
   Unit for Theologu-Carlier network analysis - node failures OK
   ---------------------------------------------------------------------
   Graph description using pointers to save stack space
   ---------------------------------------------------------------------
   Unit for graph reduction functions implementing the Theologou-Carlier
   algorithm in IEEE Transactions on Reliability, Vol 40 (June, 1991),
   pp 210-217.  Includes data structures for jamming and hidden links
   ---------------------------------------------------------------------*)
INTERFACE
USES commGraf;
TYPE
  DegreeType = ARRAY[1..MAXNODE] OF Integer;  {List of vertex degrees}
  GraphSet = SET OF 1..MAXNODE;               {Set of vertices}
  Edge = RECORD                               {Edge in a graph}
      Start,                                  {Start vertex}
      Stop : 1..MAXNODE;                      {Stop vertex }
      Beta : Real                             {Edge reliabilities}
    END; { Edge }
  EdgeSet = RECORD
      e : ARRAY[1..MAXEDGE] OF Edge;
      n : Integer;
    END; (* EdgeSet *)
```

```
  pGraph = ^Graph;
  Graph = RECORD                                  {Describes a graph}
      Vert : GraphSet;                            {Set of graph vertices}
      Source,                                     {Source vertex}
      Sink : Integer;                             {Sink vertex}
      InDegree,                                   {In degree of each vertex}
      OutDegree : DegreeType;                    {Out degree of each vertex}
      nb : ARRAY[1..MAXNODE] OF GraphSet;     {Edge(i,j) puts j in nb[i]}
      NumEdges : Integer;                   {Number of edges in the graph}
      NumNodes : Integer;          {Largest numbered vertex in the graph}
      e : ARRAY[1..MAXEDGE] OF Edge;  {Describes all edges in the graph}
      Alpha : ARRAY[1..MAXNODE] OF Real;      {Node reliabilities}
    END; { Graph }
  edgeList = ARRAY [1..MAXNODE] OF byte;
  PathMat  = ARRAY [0..MAXNODE] OF edgeList;
VAR
   pAverage : real;     { Average link success probability }
   pMax     : real;     { Maximum link success probability }
FUNCTION   Pfunction(z : Real) : Real;
FUNCTION   BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
PROCEDURE  TCSTReliabil( VAR g : Graph; VAR h: EdgeSet;
                         VAR upperRel, lowerRel : real);
(* ------------------------------------------------------------------ *)
IMPLEMENTATION
USES
  Crt;
VAR
   count : word;               { Used these                          }
   percentComplete : real;     { three variables to                  }
   modify : real;              { inform user about the progress      }
   delta : real;       { modifier of threshold }
(* ------------------------------------------------------------------ *)
FUNCTION Pfunction(z : Real) : Real;
```

[For a listing of this function, see Section D.1.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
```

[For a listing of this function, see Section D.6.2]

```
(*--------------------------------------------------------------------*)
PROCEDURE FindDegree (VAR g : Graph);
```

[For a listing of this procedure, see Section D.6.2]

```
(*--------------------------------------------------------------------*)
PROCEDURE Delete (VAR g : Graph; n : Integer);
```

[For a listing of this procedure, see Section D.6.2]

```
(*--------------------------------------------------------------------*)
PROCEDURE CleanSink (VAR g : Graph);
```

[For a listing of this procedure, see Section D.6.2]

```
(*--------------------------------------------------------------------*)
PROCEDURE CleanSource (VAR g : Graph);
```

[For a listing of this procedure, see Section D.6.2]

```
(*--------------------------------------------------------------------
*)
```

```
PROCEDURE CleanUp (VAR g : Graph);
```

[For a listing of this procedure, see Section D.6.2]

```
(*-------------------------------------------------------------------*)
PROCEDURE ForwardSimplify (VAR g : Graph; VAR Simplified : Boolean);
```

[For a listing of this procedure, see Section D.6.2]

```
(*-------------------------------------------------------------------*)
PROCEDURE BackSimplify (VAR g : Graph; VAR Simplified : Boolean);
```

[For a listing of this procedure, see Section D.6.2]

```
(*-------------------------------------------------------------------*)
PROCEDURE SourceSinkRed (VAR g:Graph; VAR Found:Boolean; VAR
Factor:Real);
```

[For a listing of this procedure, see Section D.6.2]

```
(*-------------------------------------------------------------------*)
PROCEDURE InOutDeg1Red (VAR g : Graph; VAR Found : Boolean);
```

[For a listing of this procedure, see Section D.6.2]

```
(*-------------------------------------------------------------------*)
PROCEDURE Contract (VAR g : Graph; NewSink : Integer);
```

[For a listing of this procedure, see Section D.6.2]

```
(*-------------------------------------------------------------------*)
FUNCTION Connected ( VAR g : Graph) : Boolean;
```

[For a listing of this function, see Section D.6.2]

```
(*-------------------------------------------------------------------*)
PROCEDURE SinkEdge(VAR g : Graph; VAR k : Integer; VAR InitVert :
Integer);
```

[For a listing of this procedure, see Section D.6.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION findEdge( var g : graph; sc, sk : integer) : integer;
VAR
   i : integer;
BEGIN
  WITH g DO
    FOR i := 1 to numEdges DO
      IF (e[i].start = sc) and (e[i].stop = sk) then
        BEGIN
          findEdge := i;
          exit;
        END;
END;
(* ------------------------------------------------------------------ *)
FUNCTION minP( g : Graph) : real;
VAR
  LinkMat            : PathMat;
  i, j, l, Nback     : Integer;
  result : real;
  r : boolean;
BEGIN
  WITH g DO
  BEGIN
```

```
    FOR i := 1 to maxNode DO
      LinkMat[numNodes, i] := 0;
    Move( linkMat[numNodes], linkMat[0], maxNode);
    Inc( linkMat[0, source]);
    Move( linkMat[numNodes], linkMat[1], maxNode);
    FOR j := 1 to numEdges DO         { j indexing edges }
      IF e[j].start = source THEN
         BEGIN
           linkMat[1, e[j].stop] := j;
           Inc( linkMat[0, e[j].stop]);
         END;
    l := 1;
    r := true;
    WHILE r and (linkMat[0, sink] = 0) DO
      BEGIN
        r := false;
        move( linkMat[numNodes], linkMat[l+1], maxNode);
        FOR i := 1 to maxNode DO
          IF linkMat[l, i] <> 0 THEN
            FOR j := 1 to maxNode DO
              IF (j in nb[i]) and (linkMat[0, j] = 0) THEN
                BEGIN
                  linkMat[l+1, j] := findEdge( g, i, j);
                  inc( linkMat[0, j]);
                  r := true;
                END;
        inc(l);
      END;
      IF (linkMat[0, sink] <> 0) THEN
        BEGIN
          result := 1.0;
          nBack := sink;
          FOR i := l downto 1 DO
            BEGIN
                                  result  :=  result  *  alpha[nBack]  *
e[linkMat[i,nback]].beta;
              Nback := e[linkMat[l, nBack]].start;
            END;
          minP := result * alpha[source];
        END
      ELSE minP := 0.0;
    END;
END;
(* ------------------------------------------------------------------- *)
PROCEDURE informUser;
BEGIN
  IF lo( count) = 0 then
    BEGIN
      write( 100*percentComplete:2:0, '%');
        IF (hi( count) mod 25) = 0 then
          BEGIN
            write( '                          ');
            gotoXY( whereX-29, whereY);
          END
        ELSE
          BEGIN
            gotoXY( whereX+( hi( count) mod 25), whereY);
            write( '.');
            gotoXY( whereX-4-( hi( count) mod 25), whereY);
          END;
```

```
    END;
    inc( count);
END;

(* ------------------------------------------------------------------ *)
PROCEDURE STRel( pg : pGraph; p_Thresh : real;
                 var upperBnd, lowerBnd : real);
{ Returns upper and lower bound on the reliability of the graph pg }
VAR
  Reducible : Boolean;   {True if the graph was just reduced}
  p: Real;               {Factor for probability of the reduced graph}
  MarkedEdge : Integer;  {Edge used for factoring}
  ProbEdge : Real;       {Probability of edge used for factoring}
  InitVert : Integer;    {starting point of factored edge}
  pLink : Real;
  cUpper, cLower, dUpper, dLower : real;  { contracted upper and lower
                                            and deleted upper and lower }
  alphaInit : Real;
  pLocalGraph : pGraph;
  oldModify, oldPercent : real;
BEGIN  {UpperProb}
  oldModify := modify;
  oldPercent := percentComplete;
  informUser;
  pLocalGraph := new( pGraph);
  pLocalGraph^ := pg^;        { Copy the graph into local variable }
  WITH pLocalGraph^ DO
  BEGIN
    p := 1.0;
    REPEAT
      Reducible := FALSE;
      CleanUp( pLocalGraph^ );
      IF (Source <> Sink) AND (InDegree[Sink] > 0) AND
                                (OutDegree[Source] > 0) THEN
        BEGIN
          SourceSinkRed( pLocalGraph^, Reducible, p);
          IF NOT Reducible THEN
            BEGIN  {No source or sink reduction was possible}
              BackSimplify( pLocalGraph^, Reducible);
              ForwardSimplify( pLocalGraph^, Reducible);
              InOutDeg1Red( pLocalGraph^, Reducible);
            END
        END
    UNTIL NOT Reducible;
    IF (Source = Sink) THEN
      BEGIN
        lowerBnd := p;
        UpperBnd := p;
      END
    ELSE IF (InDegree[Sink] = 0) OR (OutDegree[Source] = 0) OR
         (NOT connected( pLocalGraph^)) THEN
      BEGIN
        UpperBnd := 0;
        lowerBnd := 0;
      END
    ELSE
      BEGIN { Factor the graph -- no more reductions are possible}
        SinkEdge( pLocalGraph^, MarkedEdge, InitVert);
        alphaInit := alpha[initVert];
        ProbEdge := e[MarkedEdge].Beta;
```

```
        pLink := ProbEdge * AlphaInit;    {  Alpha of sink is always 1 }
        Delete( pLocalGraph^, MarkedEdge);
        IF (ProbEdge > P_THRESH) THEN
          BEGIN
            dLower := minP( pLocalGraph^);
            Contract(pLocalGraph^, InitVert);
            STRel( pLocalGraph, p_Thresh+delta, cUpper, cLower);
            upperBnd := p * cUpper;
            lowerBnd := p * ( pLink * cLower + (1-pLink) * dLower);
          END
        ELSE
          BEGIN
            modify := oldModify /2;
            alpha[initVert] := alphaInit * (1 - probEdge)/(1 - pLink);
            STRel( pLocalGraph, p_Thresh-delta, dUpper, dLower);
            modify := oldModify /2;
            contract( pLocalGraph^, initVert);
            STRel( pLocalGraph, p_Thresh-delta, cUpper, cLower);
            upperBnd := p * ( pLink * (cUpper - dUpper) + dUpper);
            lowerBnd := p * ( pLink * (cLower - dLower) + dLower);
          END
      END  {Factor the graph -- no more reductions are possible}
  END; (* WITH *)
    percentComplete := oldpercent + oldmodify;
    dispose( pLocalGraph);
END; {UpperProb}
(* ------------------------------------------------------------------ *)
FUNCTION HiddenProb(VAR g : Graph; VAR h: EdgeSet) : Real;
```

[For a listing of this function, see Section D.6.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION lower( r1,r2 : real) : real;
BEGIN
     IF r1 < r2 THEN
        lower := r1
     ELSE lower := r2;
END;
(* ------------------------------------------------------------------ *)
PROCEDURE TCSTReliabil( VAR g : Graph; VAR h: EdgeSet;
                        var upperRel, lowerRel : Real);
VAR
  pa, f : Real;
  tU, tL : real;
BEGIN
  WITH g DO
    BEGIN
      FindDegree(g);
      f := alpha[source] * alpha[sink];
      alpha[source] := 1;
      alpha[sink] := 1;
      pa := HiddenProb(g, h) * f;
      modify := 1.0;
      percentComplete := 0.0;
      count := 0;
      delta := (pMax - pAverage) / lower(((1.01 - pAverage) * 100), 26);
      STRel( @g, pMax, tU, tL);
      upperRel := pa + (1 - pa) * f * tU;
      lowerRel := pa + (1 - pa) * f * tL;
    END; (* WITH *)
```

```
END; (* TCSTReliabil *)
(*---------------- No Initialization ---------------------*)
END.
```

D.8 IMPLEMENTATION OF THE REDUCTION & PARTITION ALGORITHM

D.8.1 Program REDNPART.PAS and units RDNPARTU.PAS and COMMGRAF.PAS

```
{$X+}
{$S+}
{$M 65520,0,655360}
PROGRAM REDNPART;
(*   ------------------------------------------------------------
   Driver program for Theologou-Carlier network analysis unit
   ------------------------------------------------------------
   Program implementing the Reduction and Partition algorithm in IEEE
   Transactions on Reliability, Vol 41 (June, 1992), pp 201-209.
   This is a test of the RDNPARTU.PAS unit which implements the network
   with adjacency matrix and adaptive thresholds.  From jamming data
   the graph structure is calculated with its link reliabilities
   (Betas).  This is shown for any user-supplied source-sink
   selection, with an option to estimate the st-reliability for
   this selected source-sink pair. This portion requests a value
   of the node reliability (Alpha, a constant), and attempts to
   calculate upper and lower bounds on the st-reliability.
   ------------------------------------------------------------
   Note that in the unit it is not required that the node failure
   Alpha be a constant.  Alpha can very from node to node, although
   in this test program the node failure rate is a constsnt.
   ------------------------------------------------------------ *)
USES
  Crt, Dos, RDNPARTU, PopMenus, Dir_Menu,
  Strings, Keyboard, FileChck, timer, COMMGRAF;
TYPE
    SNRS = array[1..maxJam, 1..edgeMax] OF Real;
    Matrix = ARRAY[1..NODEMAX, 1..nodeMax] OF byte;
VAR
  g : Graph;
  Hidden : EdgeSet;
  Threshold, SigmaL : Real;
(* ------------------------------------------------------------------ *)
PROCEDURE BuildGraph(VAR g : Graph; VAR h : EdgeSet;
                     kj : Integer; Thresh, Sigma : Real);
```

[For a listing of this procedure, see Section D.6.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE readSNRs;
LABEL
  READ_ERROR;
VAR
  Msg, s, FileSpec, FileStr, ExtStr, SNRStr, DirSpec : String;
  SNRFile : Text;
  i, j, k : integer;
BEGIN
  FileSpec := '*.SNR';
  DirSpec := '';
```

```
  Msg := '  << SNRFile Selection (F1 for HELP)';
  SNRStr := DirectoryMenu(DirSpec, FileSpec, Msg);
  MakeStrUpper(SNRStr);
  {$V-} Fsplit(SNRStr, DirSpec, FileStr, ExtStr); {$V+}
  IF CheckOldFile(SNRFile, SNRStr) THEN
     readSNRfile( snrFile);
  Close(SNRFile);
END; (* GetSNRs *)
(* ------------------------------------------------------------------ *)
PROCEDURE GetSignalData;
```

[For a listing of this procedure, see Section D.6.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE TestSTRel(VAR g : Graph; VAR h: EdgeSet);
```

[For a listing of this procedure, see Section D.6.1]

```
(* ---------------------- Main Program ----------------------------- *)
BEGIN
  ClrScr;
  Threshold := 0.0;
  SigmaL := 10.0;
  ReadSNRs;
  GetSignalData;
  TestSTRel(g, Hidden);
END. (* Main Program *)


UNIT RDNPARTU;
(*    ------------------------------------------------------------------
   Unit for Theologu-Carlier network analysis - node failures OK
   ------------------------------------------------------------------
   Unit for graph reduction functions implementing the Reducion and
   Partition Algorithm in IEEE Transactions on Reliability, Vol 41
   (June, 1992), pp 201-209.
   ------------------------------------------------------------------*)
INTERFACE
USES COMMGRAF;
CONST
  MAXJAM = 9;           {Maximum number of jammers}
  NODEMAX = 35;          {Maximum number of nodes}
  EDGEMAX = 225;         {Maximum number of edges}
TYPE
  nodeType = byte;
  betaType = real;
  alphaType = real;
  DegreeType = ARRAY[1..NODEMAX] OF byte;     {List of vertex degrees}
  Edge = RECORD                               {Edge in a graph}
      Start,                                  {Start vertex}
      Stop : 1..NODEMAX;                      {Stop vertex }
      Beta : betaType;                        {Edge reliabilities}
    END; { Edge }
  EdgeSet = RECORD
      e : ARRAY[1..EDGEMAX] OF Edge;
      n : Integer;
    END; (* EdgeSet *)
  pGraph = ^Graph;
  Graph = RECORD                              {Describes a graph}
      net : array[1..nodeMax, 1..nodeMax] of byte;
      Source,                               {Source vertex}
```

```
      Sink : nodeType;                      {Sink vertex}
      InDegree,                             {In degree of each vertex}
      OutDegree : DegreeType;               {Out degree of each vertex}
      numNodes : byte;                     {number of nodes in the graph}
      nodeSet : array[1..nodeMax] of nodeType;
      e : ARRAY[1..EDGEMAX] OF betaType;{Beta of all edges in the graph}
      Alpha : ARRAY[1..NODEMAX] OF alphaType;   {Node reliabilities}
    END; { Graph }
   NodeList = ARRAY [1..NODEMAX] OF nodeType;
   PathMat  = ARRAY [0..NODEMAX + 1] OF NodeList;
   pathType = array [1..nodeMax] of byte;  { shortest path from s to t }
FUNCTION   Pfunction(z : Real) : Real;
FUNCTION   BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
FUNCTION   TCSTReliabil(var g : Graph; VAR h: EdgeSet) : Real;
PROCEDURE initializeGraph( var g : graph; nn : byte);
(* ------------------------------------------------------------------ *)
IMPLEMENTATION
(*    ------------------------------------------------------------------
   Most of the graph processing functions below are modifications of the
  earlier Page-Perry procedures for network reduction and factorization,
 but rewritten according to Theologu-Carlier in order to account for the
  possibility of node failures.  Adjency matrix is used to represent the
   network as oppose to the original set representation.
  ------------------------------------------------------------------*)
USES
  Crt, Dos, KeyBoard, TextScrn,
  Strings, PopMenus, FileChck;
VAR
  delta : real;
(* ------------------------------------------------------------------ *)
PROCEDURE initializeGraph( var g : graph; nn : byte);
VAR i : byte;
BEGIN
  WITH g DO
    BEGIN
      numNodes := nn;
      FOR i := 1 TO nn DO
        net[1, i] := 0;
      FOR i := 2 TO nn DO
        move( net[1], net[i], nn);
      move( net[1], inDegree, nn);
      move( net[1], outDegree, nn);
      FOR i := 1 TO nn DO
         nodeSet[i] := i;
       source := 1;
       sink := nn;
     END;
END;
(* ------------------------------------------------------------------ *)
FUNCTION Pfunction(z : Real) : Real;
```

[For a listing of this function, see Section D.1.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
```

[For a listing of this function, see Section D.6.2]

```
(*------------------------------------------------------------------*)
```

```
PROCEDURE Delete (VAR g : Graph; sv, tv : nodeType);
BEGIN
  WITH g DO
  BEGIN
    Net[sv,tv] := 0;
    Dec(inDegree[tv]);
    Dec(outDegree[sv]);
  END; (* WITH *)
END;
(*-------------------------------------------------------------------------*)
PROCEDURE CleanSink (VAR g : Graph);
VAR
  i : byte;
BEGIN
  i := 0;
  WITH g DO
    WHILE outDegree[sink] > 0 DO
    BEGIN
      REPEAT
        Inc(i)
      UNTIL net[sink, nodeSet[i]] <> 0;
      Delete(g, sink, nodeSet[i]);
    END;
END; (* CleanSink *)
(*-------------------------------------------------------------------------*)
PROCEDURE CleanSource (VAR g : Graph);
VAR
  j : byte;
BEGIN
  j := 0;
  WITH g DO
    WHILE inDegree[source] > 0 DO
    BEGIN
      REPEAT
        Inc(j)
      UNTIL net[nodeSet[j], Source] <> 0;
      Delete(g, nodeSet[j], source);
    END;
END; (* CleanSource *)
(* ------------------------------------------------------------------------ *)
PROCEDURE eleminateNode( var g : graph; n : nodeType);
          { make sure there is no link either in or
            out of this node before calling this procedure }
VAR i : byte;
BEGIN
  WITH g DO
  BEGIN
    i := 0;
    REPEAT
      inc(i);
    UNTIL nodeSet[i] = n;
    move( nodeSet[i+1], nodeSet[i], numNodes-i);
    dec(numNodes);
  END;
END;
(*-------------------------------------------------------------------------*)
PROCEDURE CleanUp (VAR g : Graph);
VAR
  i, j, k, l : nodeType;
 Reduced : Boolean; {Set false if dead end or false start vertex found}
```

```
BEGIN
  CleanSource(g);
  CleanSink(g);
  WITH g DO
    REPEAT
      Reduced := TRUE;
      FOR k := numNodes DOWNTO 1 DO
        BEGIN
          j := nodeSet[k];
          IF (InDegree[j] = 0) OR (OutDegree[j] = 0) THEN
            IF (j <> Source) AND (j <> Sink) THEN
              BEGIN
                FOR l := 1 TO numNodes DO
                  BEGIN
                    i := nodeSet[l];
                    IF net[j,j] <> 0 THEN delete(g, j, i);
                  END;
                eleminatenode(g, j);
                Reduced := FALSE;
         END;
    UNTIL Reduced
END;
(*-------------------------------------------------------------------------*)
PROCEDURE ForwardSimplify (VAR g : Graph; VAR Simplified : Boolean);
{ If one exists, eliminates a nonnecessary edge coming
  into a vertex and sets Simplified to TRUE.}
VAR
  i, j : nodeType;
  v : nodeType;  {Initial vertex for an edge}
BEGIN {ForwardSimplify}
  WITH g DO
    FOR j := 1 TO numNodes DO
      IF (OutDegree[nodeSet[j]] = 1) THEN
        BEGIN
          v := nodeSet[j];
          FOR i := 1 TO numNodes DO
         IF (net[v, nodeSet[i]] <> 0) and (net[nodeSet[i], v] <> 0) THEN
              BEGIN
                delete( g, nodeSet[i], v);
                Simplified := TRUE;
              END;
           END;  {Delete the antiparallel edge.}
END;  {ForwardSimplify}
(*-------------------------------------------------------------------------*)
PROCEDURE BackSimplify (VAR g : Graph; VAR Simplified : Boolean);
{ If one exists, eliminates a nonnecessary edge coming
  out of a vertex and sets Simplified to TRUE.}
VAR
  i, j : nodeType;
  v : nodeType;  {Terminal vertex for an edge}
BEGIN  {BackSimplify};
  WITH g DO
    FOR j := 1 TO numNodes DO
      IF (InDegree[nodeSet[j]] = 1) THEN
        BEGIN
          v := nodeSet[j];
          FOR i := 1 TO numNodes DO
               IF (net[nodeSet[i],v] <> 0) AND (net[v,nodeSet[i]] <> 0)
THEN
                BEGIN
```

```
                delete( g, v, nodeSet[i]);
                Simplified := TRUE;
              END;
        END;  {Delete the antiparallel edge.}
END;  {BackSimplify}
(*-------------------------------------------------------------------------*)
PROCEDURE SourceSinkRed( VAR g : Graph; VAR Found : Boolean;
                         VAR Factor : Real);
{ If the sink of graph g has in-degree 1, then it is merged into its
 neighbor and the resulting sink is cleaned of out-edges.  If the source
  has out-degree 1, then the parallel result occurs.
  Alpha of the original source or sink are assumed to be 1.0.  Factor is
  returned as the appropriate multiplying factor for the graph.}
VAR
  j : nodeType;            {Possible neighbor of  source or sink}
  OldNode :   nodeType;    {Original source or sink vertex}
BEGIN  {SourceSinkRed}
  WITH g DO
    BEGIN
      IF InDegree[Sink] = 1 THEN
        BEGIN  {Merge the sink Into its adjacent vertex.}
          Found := TRUE;
          OldNode := Sink;
          FOR j := 1 TO numNodes DO
            IF (net[nodeSet[j], oldNode] <> 0) THEN Sink := nodeSet[j];
          Factor := Factor * e[net[sink,oldNode]] * alpha[Sink];
          alpha[sink] := 1.0;
          Delete(g, sink, oldNode);
          eleminateNode( g, oldNode);
          CleanSink(g);
        END;  {Merge the sink into its adjacent vertex.}
      IF (OutDegree[Source] = 1) AND (Source <> Sink) THEN
        BEGIN  {Merge the source Into its adjacent vertex.}
          Found := TRUE;
          OldNode := Source;
          FOR j := 1 TO numNodes DO
            IF (net[oldNode,nodeSet[j]] <> 0) THEN Source := nodeSet[j];
          Factor := e[net[oldNode,source]] * Factor * Alpha[Source];
          alpha[Source] := 1.0;
          Delete(g, oldNode, Source);
          eleminateNode( g, oldNode);
          CleanSource(g);
        END;  {Merge the source into its adjacent vertex.}
  END; (* WITH *)
END; {SourceSinkRed}
(*-------------------------------------------------------------------------*)
PROCEDURE InOutDeg1Red (VAR g : Graph; VAR Found : Boolean);
{ G is scanned to find a vertex with in-degree and out-degree 1. If such
  a vertex is found, it it removed and the resulting graph simplified.}
VAR
  i, j : nodeType;         {Graph edge}
  u : nodeType;          {Graph vertex (with possible in/out degree 1)}
  edgeNumber : byte;     {edge number of one of the elimiated edge}
  DoubleRel : Real;      {Reliability of both edges in sequence}
  InitV : nodeType;      {Initial vertex with edge into u}
  TermV : nodeType;      {Terminal vertex with edge out of u}
BEGIN {InOutDeg1Red}
  WITH g DO
    FOR j := 1 TO numNodes DO
      IF (InDegree[nodeSet[j]] = 1) AND (OutDegree[nodeSet[j]] = 1) THEN
```

```
        BEGIN  {Vertex u has in- and out-degree 1.  Eliminate it.}
          u := nodeSet[j];
          Found := TRUE;
          FOR i := 1 TO numNodes DO
            IF net[nodeSet[i], u] <> 0 THEN initV := nodeSet[i];
          FOR i := 1 TO numNodes DO
            IF net[u, nodeSet[i]] <> 0 THEN termV := nodeSet[i];
          edgeNumber := net[initV, u];
          doubleRel := e[edgeNumber] * e[net[u,termV]]* alpha[u];
          Delete(g, initV, u);
          delete(g, u, termV);
          IF net[initV, termV] <> 0 THEN
               e[net[initV,termV]] := e[net[initV,termV]] *
                                          (1 - DoubleRel) + DoubleRel
          ELSE
            BEGIN  {Reuse elimated edge to Construct a new edge}
              net[initV, termV] := edgeNumber;
              e[edgeNumber] := doubleRel;
              Inc(InDegree[TermV]);
              Inc(OutDegree[InitV]);
            END;  {Construct a new edge from InitV to TermV}
          eleminateNode( g, u);
          Exit;
        END; {Vertex u has in- and out-degree 1.  Eliminate it.}
END; {DegTwoRed}
(*-------------------------------------------------------------------------*)
PROCEDURE ContractToSource (VAR g : Graph; newSource : nodeType);
{ Contracts the sink of graph g into the vertex NewSink.}
VAR
  u, v, oldSource : nodeType;        {Graph vertex}
BEGIN {Contract}
  WITH g DO
    BEGIN
      u := 0;
      WHILE outDegree[source] <> 0 DO
      BEGIN
        REPEAT
          inc(u)
        UNTIL net[source, nodeSet[u]] <> 0;
        v := nodeSet[u];
        IF net[newSource,v] <> 0 THEN   { Parallel edge }
                  e[net[NewSource,v]] := e[net[newSource,v]]
                                         * (1 - e[net[Source,v]])
                                         + e[net[Source,v]]
        ELSE
          BEGIN { Change edge outSource}
            net[newSource,v] := net[Source,v];
            Inc(outDegree[NewSource]);
            inc(inDegree[v]);
          END;
        delete( g, source, v);
      END;
      oldSource := source;
      Source := NewSource;
      eleminateNode(g, oldSource);
      CleanSource(g);
  END; (* WITH *)
END;  {Contract}
(*-------------------------------------------------------------------------
*)
```

```
PROCEDURE PATH3( VAR g : Graph; VAR r : boolean; VAR pth : pathType;
                 var l : integer);
(* If st-path is found, set up the path links used and return TRUE  *)
(* Use forward flood searches to find path--no backward search used *)
(* use linkMat[0] to keep track of the visited nodes so the procedure *)
(*will not stuck in a loop                                           *)
VAR
  LinkMat         : PathMat;
  lastNode        : nodeType;
  i, j, Nback     : Integer;
BEGIN
  WITH g DO
    BEGIN
      lastNode := nodeSet[numNodes];
      FOR j := 1 TO lastNode DO LinkMat[nodeMax, j] := 0;
      move( linkMat[nodeMax], linkMat[0], lastNode);
      linkMat[0, source] := 1;
      move( linkMat[nodeMax], linkMat[1], lastNode);
      FOR j := 1 TO numNodes DO
        IF net[source,nodeSet[j]] <> 0 THEN
          BEGIN
            linkMat[1, nodeSet[j]] := source;
            inc(linkMat[0, nodeSet[j]]);
          END;
      l := 1;
      r := true;
      WHILE r AND (linkMat[0, sink] = 0) do
        BEGIN
          r := false;
          move( linkMat[nodeMax], linkMat[l+1], lastNode);
          FOR i := 1 TO lastNode DO
            IF linkMat[l, i] <> 0 THEN
              FOR j := 1 TO numNodes DO
                IF (linkMat[0, nodeSet[j]] = 0)
                          AND (net[i,nodeSet[j]] <> 0) THEN
                  BEGIN
                    linkMat[l+1, nodeSet[j]] := i;
                    inc( linkMat[0, nodeSet[j]]);
                    r := true;
                  END;
          inc(l);
        END;
        r := (linkMat[0, sink] <> 0);
        IF r  THEN
          BEGIN
            nBack := sink;
            FOR i := 1 TO l DO
              BEGIN
                pth[l-i+1] := nBack;
                Nback := linkMat[l-i+1, nBack];
              END;
          END;
     END;
END;
(* ------------------------------------------------------------------ *)
FUNCTION minP( var g : graph) : real;
VAR
  i : integer;
  found : boolean;
  path : pathType;
```

```
  r : real;
BEGIN
  path3( g, found, path, i);
  IF found THEN WITH g DO
    BEGIN
      r := 1.0;
      WHILE i > 1 DO
        BEGIN
          r := alpha[path[i]] * e[net[path[i-1], path[i]]];
          dec(i);
        END;
      r := r * alpha[source]*e[net[source, path[1]]];
      minP := r;
    END
  ELSE minP := 0;
END;
(* ---------------------------------------------------------------------- *)
FUNCTION RedNPart(pg : pGraph) : Real;
VAR
  Reducible : Boolean;   {True if the graph was just reduced}
  p : Real;              {Factor for probability of the reduced graph}
  ProbEdge : Real;       {Probability of edge used for factoring}
  endVert : nodeType;     {Endpoint of factored edge}
  r : Real;              {Probability of g with edge removed}
  pLink : Real;
  pLocalGraph : pGraph;
  found : boolean;
  path : pathType;
  i, length : integer;
  temp : real;
BEGIN
     pLocalGraph := new( pGraph);
     pLocalGraph^ := pg^;        { Copy the graph into local variable }
     WITH pLocalGraph^ DO
     BEGIN
          p := 1.0;
          REPEAT
                Reducible := FALSE;
                CleanUp( pLocalGraph^ );
                IF (Source <> Sink) AND (InDegree[Sink] > 0) AND
                   (OutDegree[Source] > 0) THEN
                BEGIN
                     SourceSinkRed( pLocalGraph^, Reducible, p);
                     IF NOT Reducible THEN
                     BEGIN  {No source or sink reduction was possible}
                            BackSimplify( pLocalGraph^, Reducible);
                            ForwardSimplify( pLocalGraph^, Reducible);
                            InOutDeg1Red( pLocalGraph^, Reducible);
                     END
                END
          UNTIL NOT Reducible;
          IF (Source = Sink) THEN
             redNPart := p
          ELSE
            BEGIN
              Path3( pLocalGraph^, found, path, length);
              IF found THEN
                BEGIN
                  r := 0.0;
                  FOR i := 1 TO length DO
```

```
                    BEGIN
                      endVert := path[i];
                      ProbEdge := e[net[source,endVert]];
                      pLink := ProbEdge * alpha[endVert];
                      Delete( pLocalGraph^, source, endVert);
                      temp := p * pLink;
                      IF pLink < 1.0 THEN
                         alpha[endVert] := (alpha[endVert] - pLink)
                                              / (1 - pLink);
                      r := r + (p - temp) * redNPart( pLocalGraph);
                      p := temp;
                      IF i < length THEN
                         BEGIN
                           contractToSource( pLocalGraph^, endVert);
                           alpha[source] := 1.0;
                         END;
                    END;
                    redNPart := r + p;
                 END
              ELSE redNPart := 0.0;
           END;
    END; (* WITH *)
    dispose( pLocalGraph);
END;
(* ------------------------------------------------------------------ *)
FUNCTION HiddenProb(VAR g : Graph; VAR h: EdgeSet) : Real;
```

[For a listing of this function, see Section D.6.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION TCSTReliabil(var g : Graph; VAR h: EdgeSet) : Real;
VAR
  p : Real;
BEGIN
  WITH g DO
    BEGIN
      p := Alpha[Source] * Alpha[Sink];
      alpha[source] := 1.0;
      alpha[sink] := 1.0;
      TCSTReliabil := p * redNPart( @g);
    END; (* WITH *)
END;
(*---------------- No Initialization ---------------------*)
END.


UNIT commGraf;
                                                                    {
*****************************************************************
           This unit implements a 'standard' SNR network by reading in
          the data in an SNR file.  The following functions pertinent to
          the SNR network are provided.
     PROCEDURE initializeSNRs;
               This procedure allocates memory for the SNR network and
               initialize the network to empty.
     PROCEDURE readSNRFile( var snrFile : text);
               This procedure fills in the network with the data
               in the file snrFile.  Note: snrFile is not a string but an
               opened text file.
     FUNCTION getSNRValue( sourceNode, terminalNode : integer;
```

```
                    jamCase: integer) : real;
              This function returns the SNR value from sourceNode to
              terminalNode with jamCase active;
     FUNCTION getNumNodes : integer;
              This function returns the number of nodes in the network.
     FUNCTION getNumJams : integer;
             This function returns the number of jammers in the network.
     PROCEDURE clearSNRS;
             This procedure deallocates the memory and it should be
             called before the program terminates.
   ************************************************************** }
INTERFACE
USES dos;
CONST
     maxNode = 35;
     maxJam  = 9;
     maxEdge = 255;
TYPE
    graphMat = array[1..maxNode, 1..maxNode] of byte;
    edges = array[1..maxEdge] of real;
VAR
   ioError : boolean;
PROCEDURE initializeSNRs;
PROCEDURE readSNRFile( var snrFile : text);
FUNCTION getSNRValue( sourceNode, terminalNode : integer;
                      jamCase: integer) : real;
FUNCTION getNumNodes : integer;
FUNCTION getNumJams : integer;
PROCEDURE clearSNRS;

IMPLEMENTATION
TYPE
    snrMat = array[1..maxJam] of edges;
    pSnrMat = ^snrMat;
    pGrafMat = ^graphMat;
VAR
   nodeNum, jamNum : byte;
   g : pGrafMat;
   e : pSnrMat;
   initialized : boolean;

PROCEDURE initializeSNRs;
VAR i : byte;
BEGIN
     IF NOT initialized THEN
     BEGIN
          g := new(pGrafMat);
          e := new(pSnrMat);
          for i := 1 to maxNode do
              g^[1, i] := 0;
          for i := 2 to maxNode do
              move( g^[1], g^[i], maxNode);
          nodeNum := 0;
          jamNum := 0;
          initialized := true;
     END;
END;

PROCEDURE readSNRFile( var snrFile : text);
VAR i, j, k : integer;
```

```
    s : string;
    edgeCount : byte;
BEGIN
     initializeSNRS;
     {$I-} Readln(SNRFile, s); {$I+}
     IF (IOResult = 0) AND (ExitCode = 0) THEN
     BEGIN
          {$I-} Readln(SNRFile, i, nodeNum, jamNum); {$I+}
          initialized := false;
          edgeCount := 0;
          WHILE NOT EOF(SNRFile) DO
          BEGIN
               inc(edgeCount);
               {$I-} Read(SNRFile, i, j); {$I+}
               G^[i,j] := edgeCount;
               FOR k := 1 to jamNum DO
                   {$I-} Read(SNRFile, e^[k, edgeCount]); {$I+}
               {$I-} Readln(SNRFILE); {$I+}
          END; (* WHILE *)
     END ELSE ioError := true;
END;

FUNCTION getSNRValue( sourceNode, terminalNode : integer;
                      jamCase: integer) : real;
CONST bigNeg = -99.9;
BEGIN
     IF g^[sourceNode, terminalNode] <> 0 THEN
        getSNRValue := e^[jamCase, g^[sourceNode, terminalNode]]
     else getSNRValue := bigNeg;
END;

FUNCTION getNumNodes : integer;
BEGIN
     getNumNodes := nodeNum;
END;

FUNCTION getNumJams : integer;
BEGIN
     getNumJams := jamNum;
END;

PROCEDURE clearSNRS;
BEGIN
     IF initialized THEN
     BEGIN
          dispose(g);
          dispose(e);
          initialized := false;
     END;
END;
BEGIN
     initialized := false;
     initializeSNRS;
END.
```

D.8.2 **Program RNPBOUND.PAS and unit RNPULUNT.PAS**

```
{$X+}
{$S+}
{$M 65520,0,655360}
PROGRAM RNPBOUND;
(*   ------------------------------------------------------------
   Driver program for Theologou-Carlier network analysis unit
   ------------------------------------------------------------
   Program implementing the Reduction and Partition algorithm in IEEE
   Transactions on Reliability, Vol 41 (June, 1992), pp 201-209.
   This is a test of the RNPULUNT.PAS unit which implement the network
   with adjacency matrix and calculate the st-Reliability to within an
   upper and lower bound.
   ------------------------------------------------------------
   Note that in the unit it is not required that the node failure
   Alpha be a constant.  Alpha can very from node to node, although
   in this test program the node failure rate is a constsnt.
   ------------------------------------------------------------ *)
USES
  Crt, Dos, rnpulunt, commGraf, PopMenus, Dir_Menu,
  Strings, Keyboard, FileChck, timer;
VAR
  g : Graph;
  Hidden : EdgeSet;
  Threshold, SigmaL : Real;
(* ------------------------------------------------------------------ *)
PROCEDURE BuildGraph(VAR g : Graph; VAR h : EdgeSet;
                     kj : integer);
```

[For a listing of this procedure, see Section D.6.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE ReadTripleData;
```

[For a listing of this procedure, see Section D.6.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE GetSignalData;
```

[For a listing of this procedure, see Section D.6.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE TestSTRel(VAR g : Graph; VAR h: EdgeSet);
```

[For a listing of this procedure, see Section D.6.1]

```
(* ---------------------- Main Program ----------------------------- *)
BEGIN
  setcBreak( true);
  ClrScr;
  Threshold := 0.0;
  SigmaL := 10.0;
  ReadTripleData;
  GetSignalData;
  TestSTRel(g, Hidden);
END. (* Main Program *)



UNIT rnpulunt;
```

```
(*    ---------------------------------------------------------------------
   Unit for Theologu-Carlier network analysis - node failures OK
   ---------------------------------------------------------------------
   Unit for graph reduction functions implementing the Reduction and
   Partition Algorithm in IEEE Transactions on Reliability, Vol 41
   (June, 1992), pp 201-209.
   ---------------------------------------------------------------------*)
INTERFACE
USES commGraf;
VAR
   pMax, pAverage : real;
TYPE
  nodeType = byte;
  betaType = real;
  alphaType = real;
  DegreeType = ARRAY[1..MAXNODE] OF byte;     {List of vertex degrees}
  Edge = RECORD                               {Edge in a graph}
      Start,                                  {Start vertex}
      Stop : 1..MAXNODE;                      {Stop vertex }
      Beta : betaType;                        {Edge reliabilities}
    END; { Edge }
  EdgeSet = RECORD
      e : ARRAY[1..MAXEDGE] OF Edge;
      n : Integer;
    END; (* EdgeSet *)
  pGraph = ^Graph;
  Graph = RECORD                              {Describes a graph}
      net : array[1..MaxNode, 1..MaxNode] of byte;
      Source,                                 {Source vertex}
      Sink : nodeType;                        {Sink vertex}
      InDegree,                               {In degree of each vertex}
      OutDegree : DegreeType;                 {Out degree of each vertex}
      numNodes : byte;                       {number of nodes in the graph}
      nodeSet : array[1..MaxNode] of nodeType;
     e : ARRAY[1..MAXEDGE] OF betaType; {Beta of all edges in the graph}
      Alpha : ARRAY[1..MAXNODE] OF alphaType;   {Node reliabilities}
    END; { Graph }

   NodeList = ARRAY [1..MAXNODE] OF nodeType;
   PathMat  = ARRAY [0..MAXNODE + 1] OF NodeList;
   pathType = array [1..MaxNode] of byte;    { shortest path from s to t
}
FUNCTION   Pfunction(z : Real) : Real;
FUNCTION   BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
PROCEDURE TCSTReliabil(VAR g : Graph; var h : edgeSet; var UB, LB :
Real);
PROCEDURE initializeGraph( var g : graph; nn : byte);
(* ------------------------------------------------------------------ *)
IMPLEMENTATION
(*   ---------------------------------------------------------------------
   Most of the graph processing functions below are modifications of the
       earlier  Page-Perry  procedures  for  network  reduction  and
factorization,
    but rewritten according to Theologu-Carlier in order to account for
the
    possibility of node failures.  Adjency matrix is used to represent
the
   network as oppose to the original set representation.
   The recursion trunction process applies when the threshold value
   P_THRESH (set to 0.9 above) is exceeded.
```

```
    Data structures for jamming and hidden links are used in the
calculation.
   ----------------------------------------------------------------*)
USES
  Crt, Dos, KeyBoard, TextScrn, Strings, PopMenus, FileChck;
VAR
   delta : real;
(* ------------------------------------------------------------------ *)
PROCEDURE initializeGraph( var g : graph; nn : byte);
```

[For a listing of this procedure, see Section D.8.1]

```
(* ------------------------------------------------------------------ *)
FUNCTION Pfunction(z : Real) : Real;
```

[For a listing of this function, see Section D.1.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION BetaQ(RhodB, MargindB, SigmadB : Real) : Real;
```

[For a listing of this function, see Section D.6.2]

```
(*----------------------------------------------------------------*)
PROCEDURE Delete (VAR g : Graph; sv, tv : nodeType);
```

[For a listing of this procedure, see Section D.8.1]

```
(*----------------------------------------------------------------*)
PROCEDURE CleanSink (VAR g : Graph);
```

[For a listing of this procedure, see Section D.8.1]

```
(*----------------------------------------------------------------*)
PROCEDURE CleanSource (VAR g : Graph);
```

[For a listing of this procedure, see Section D.8.1]

```
(* ------------------------------------------------------------------ *)
PROCEDURE eleminateNode( var g : graph; n : nodeType);
```

[For a listing of this procedure, see Section D.8.1]

```
(*----------------------------------------------------------------*)
PROCEDURE CleanUp (VAR g : Graph);
```

[For a listing of this procedure, see Section D.8.1]

```
(*----------------------------------------------------------------*)
PROCEDURE ForwardSimplify (VAR g : Graph; VAR Simplified : Boolean);
```

[For a listing of this procedure, see Section D.8.1]

```
(*----------------------------------------------------------------*)
PROCEDURE BackSimplify (VAR g : Graph; VAR Simplified : Boolean);
```

[For a listing of this procedure, see Section D.8.1]

```
(*----------------------------------------------------------------*)
PROCEDURE SourceSinkRed( VAR g : Graph; VAR Found : Boolean;
```

[For a listing of this procedure, see Section D.8.1]

```
(*----------------------------------------------------------------*)
PROCEDURE InOutDeg1Red (VAR g : Graph; VAR Found : Boolean);
```

[For a listing of this procedure, see Section D.8.1]

```
(*-------------------------------------------------------------------------*)
PROCEDURE ContractToSource (VAR g : Graph; newSource : nodeType);
```

[For a listing of this procedure, see Section D.8.1]

```
(*-------------------------------------------------------------------------*)
PROCEDURE PATH3( VAR g : Graph; VAR r : boolean; VAR pth : pathType;
                 var l : integer);
```

[For a listing of this procedure, see Section D.8.1]

```
(* ---------------------------------------------------------------------- *)
FUNCTION minP( var g : graph) : real;
VAR i : integer;
    found : boolean;
    path : pathType;
    r : real;
BEGIN
     path3( g, found, path, i);
     IF found THEN
        WITH g DO
        BEGIN
          r := 1.0;
          WHILE i > 1 DO
            BEGIN
              r := alpha[path[i]] * e[net[path[i-1], path[i]]];
              dec(i);
            END;
          r := r * alpha[source]*e[net[source, path[1]]];
          minP := r;
        END
      ELSE minP := 0;
END;
(* ---------------------------------------------------------------------- *)
PROCEDURE RedNPartUL( pg : pGraph; pThresh : real;
                    var lower, upper : real);
VAR
  Reducible : Boolean;   {True if the graph was just reduced}
   p, pL, pU : Real;            {Factor for probability of the reduced
graph}
  ProbEdge : Real;       {Probability of edge used for factoring}
  endVert : nodeType;     {Endpoint of factored edge}
  rL, rU : Real;              {Probability of g with edge removed}
  pLink : Real;
  pLocalGraph : pGraph;
  found : boolean;
  path : pathType;
  i, length : integer;
  tl, tu : real;
  tempL, tempU : real;
BEGIN
     pLocalGraph := new( pGraph);
     pLocalGraph^ := pg^;        { Copy the graph into local variable }
     WITH pLocalGraph^ DO
     BEGIN
          p := 1.0;
          REPEAT
                Reducible := FALSE;
                CleanUp( pLocalGraph^ );
                IF (Source <> Sink) AND (InDegree[Sink] > 0) AND
                   (OutDegree[Source] > 0) THEN
```

```
            BEGIN
                 SourceSinkRed( pLocalGraph^, Reducible, p);
                 IF NOT Reducible THEN
                 BEGIN  { No source or sink reduction was possible }
                      BackSimplify( pLocalGraph^, Reducible);
                      ForwardSimplify( pLocalGraph^, Reducible);
                      InOutDeg1Red( pLocalGraph^, Reducible);
                 END
            END
        UNTIL NOT Reducible;
        IF (Source = Sink) THEN
        BEGIN
          upper := p;
          lower := p;
        END ELSE
        BEGIN
          Path3( pLocalGraph^, found, path, length);
          IF found THEN
            BEGIN
              rL := 0.0;
              rU := 0.0;
              pL := p;
              pU := p;
              FOR i := 1 TO length DO
                BEGIN
                  endVert := path[i];
                  ProbEdge := e[net[source,endVert]];
                  pLink := ProbEdge * alpha[endVert];
                  Delete( pLocalGraph^, source, endVert);
                  tl := pL * pLink;
                  IF (probEdge <= pThresh) { or ( i = 1)} THEN
                    BEGIN
                      pThresh := pThresh - delta;
                      tu := pU * pLink;
                      alpha[endVert] := (alpha[endVert] - pLink)
                                              / (1 - pLink);
                      redNPartUL( pLocalGraph, pThresh, tempL, tempU);
                      rL := rL + (pL - tl) * tempL;
                      rU := rU + (pU - tu) * tempU;
                      pU := tu;
                    END ELSE
                    BEGIN
                      rL := rL + (pL - tL) * minP( pLocalGraph^);
                      pThresh := pThresh + delta;
                    END;
                  pl := tl;
                  IF i <> length THEN
                    contractToSource( pLocalGraph^, endVert);
                  alpha[source] := 1.0;
                END;
                upper := ru + pu;
                lower := rl + pl;
            END ELSE
            BEGIN
              upper := 0.0;
              lower := 0.0;
            END;
        END;
    END; (* WITH *)
    dispose( pLocalGraph);
```

```
END;
(* ------------------------------------------------------------------ *)
FUNCTION HiddenProb(VAR g : Graph; VAR h: EdgeSet) : Real;
```

[For a listing of this procedure, see Section D.6.2]

```
(* ------------------------------------------------------------------ *)
FUNCTION lower( r1 , r2 : real) : real;
BEGIN
  IF r1 > r2 THEN lower := r2
  ELSE lower := r1;
END;
(* ------------------------------------------------------------------ *)
PROCEDURE TCSTReliabil(VAR g : Graph; var h : edgeSet; var UB, LB :
Real);
VAR
  p : Real;
BEGIN
     delta := (pMax - pAverage) / lower(((1.01 - pAverage) * 100), 26);
  WITH g DO
    BEGIN
      p := Alpha[Source] * Alpha[Sink];
      alpha[source] := 1.0;
      alpha[sink] := 1.0;
      redNPartUL( @g, pMax, LB, UB);
      UB := p * UB;
      LB := p * LB;
    END; (* WITH *)
END;
(*---------------- No Initialization ----------------------*)
END.
```

APPENDIX E
ALGORITHM TESTS

In this appendix, certain *s-t* reliability algorithm tests are summarized, leading to development of the baseline computer programs used for comparison in Section 2.3.

E.1 PRELIMINARY TESTS

The algorithm performance results presented in this subsection were used to establish the basic rankings of the various programs implementing the algorithms. In Section E.2, efforts to optimize the programs are documented.

E.1.1 Results for the 3 × 3 Example Network ($\gamma_{12} = 0.93133093$)

The first set of results for the 3 × 3 example network, given below in Table E-1, are from runs made on a 286-class desktop computer with a 16 MHz clock, a coprocessor, and a SCSI hard drive. Execution times include the initialization of the network data.

TABLE E-1 FIRST SET OF RESULTS FOR 3 × 3 EXAMPLE

Program	LB	UB	Deviation	Time (sec)
EQLNKTST	.93133093 (11 successes)	.93133093 (27 failures)	0.00	3.0
EL1&2	.93133093 (11 successes)	.93133093 (19 failures)	0.00	6.7
EL2ONLY	.93133093 (14 successes)	.93133093 (19 failures)	0.00	4.1
ELCUTPAT	.93133093 (11 successes)	.93133093 (19 failures)	0.00	4.9

Obviously, for all the programs exact calculations were made, since the lower and upper bounds are equal. The reason for this behavior is that the stopping criterion forced the programs to complete the calculation. Note that the numbers of success and failure events found by the various implementations agree with the manual examples in Appendices A, B, and C.

The relative times of the programs seem reasonable in view of the operations performed. Evidently, the ELA2 cutset method (EL2ONLY), which finds a total of 33 events, is slower than the orginal ELA (EQLNKTST), which finds a greater total of 38

events. The EL1&2 program not surprisingly takes about as long as the sum of the ELA and ELA2, while the more efficient ELCUTPATH hybrid of ELA and ELA2 techniques takes only slightly more time than the ELA2.

The second set of results for the 3×3 example network, given below in Table E-2, are from runs made on a 386-class desktop computer with a 25 MHz clock. Execution times do not include the initialization of the network data.

TABLE E-2 SECOND SET OF RESULTS FOR 3×3 EXAMPLE

Program	LB	UB	Deviation	Time (sec)
EQLNKTST	.93133093	.93133093	0.00	1.6
EL1&2	.93133093	.93133093	0.00	4.0
EL2ONLY	.93133093	.93133093	0.00	2.3
ELCUTPAT	.93133093	.93133093	0.00	2.7
TCPTRBND	.93133093	.93714588	0.0029	0.1
RNPBOUND	.93133093	.93133093	0.00	< 0.05

It is obvious from these results that the four equivalent-links programs as a class ran longer than the two programs using reduction techniques. This difference is attributable in part to the fact that the equivalent-links programs are constantly accessing the hard disk to perform the queueing, while the reduction programs do not have an overhead associated with hard disk input/output.

For the program RNPBOUND, the heuristic probability threshold

$$p_{th} = \overline{\beta} + 0.75(\beta_{max} - \overline{\beta}) \tag{E-1a}$$

$$= 0.84157486 \tag{E-1b}$$

was used for both upper and lower bounds in a manner analogous to that for TCPTR-BND, where $\overline{\beta}$ is the mean value of the viable link reliabilities and β_{max} is their maximum value. The program TCPTRBND used fixed probability thresholds of 0.9 and 0.96 to obtain upper and lower bounds, respectively.

The selection of an algorithm cannot be based on the comparison expressed in these numbers because this particular example does not test the truncation and convergence features of the algorithms that will become critical for larger networks.

E.1.2 **Results for the 15-node Example Network ($\gamma_{8,13} = 0.92982718$)**

The results for the 15-node example network, given below in Table E-3, are from runs made on a 386-class desktop computer with a 25 MHz clock. It is evident from the results for the program EL2ONLY, at least for this example network, that the preference given to finding cutsets and failure events in order to accumulate an upper bound delays the accumulation of the lower bound significantly. The program ELCUTPAT, while also seeking cutsets gives preference to finding short paths, thus accumulating the lower bound faster than ELA2.

TABLE E-3 RESULTS FOR 15-NODE EXAMPLE

Program	LB	UB	Deviation	Time (sec)
EQLNKTST	.92067471 (163 successes)	.93065254 (11 failures)	0.0042	4.5
EL1&2	.92035873 (160 successes)	.93024136 (549 failures)	0.0045	23.5
EL2ONLY[14]	----	-----	----	$\gg$ 4 minutes
ELCUTPAT	.92132948 (449 successes)	.93010540 (70 failures)	0.0041	18.0
TCPTRBND	.90847146	.93039147	0.0104	23.8
RNPBOUND	.92942129	.92997859	0.0001	2.2

Note that ranking of the equivalent-links programs in terms of speed, except for the elimination of EL2ONLY, is the same as for the previous examples. However, there is a significant difference in the performances of the two recursive algorithms: these programs were an order of magnitude faster than the equivalent-links programs for the previous two examples of relatively small networks, but for the 15-node example network their speed is of the same order of magnitude. The fact that TCPTRBND took longer than EQLNKTST for this example (and more than an order of magnitude longer than RNPBOUND) is due to the fact that few of the link reliabilities exceed the fixed probability thresholds that were used, causing the program to calculate more recursions.

[14]On the 286-class computer, EL2ONLY after running for over four minutes had accumulated a lower bound of 0.665605483 (2140 successes) and an upper bound of 0.93005528 (4258 failures).

Note from Table E-1 that the exact calculation for this example took the TCPTR program 49 seconds.

E.1.3 **Results for the 34-node Example Network**

The results for the 34-node example network, given below in Tables E-4 to E-6, are from runs made on a 386-class desktop computer with a 25 MHz clock. For the equivalent-links programs, a 10-hop path limit was used. The only conclusive observations that can be made using the data in these tables is that the EL2ONLY program is definitely too time-consuming to be considered for calculation of the s-t reliabilities of large networks, and that the recursive algorithms definitely should be considered. The link reliabilities in these examples generally are too high (yielding $\gamma_{st} > 0.99$) to give an adequate picture of the relative performances of the algorithms. The stopping criterion of $\mathrm{UB} - \mathrm{LB} < 0.01$ for the equivalent-links programs obviously does not operate in the intended manner when $\gamma_{st} > 0.99$, and it appears that the criterion for stopping was the requirement to process 100 events, in the case of node pairs (3, 11) and (18, 25), while for node pair (25, 20) the algorithms stopped when the lower bound exceeded 0.99. Interestingly, the most accurate program for (25, 20) for this case was RNPBOUND, which features an adaptive probability threshold.

In order to obtain a better comparison of the algorithm performances for large networks, additional tests were conducted using the data for the 15- and 34-node example networks, modified to simulate the degradation of the link SNRs in 1 dB steps. The results of those tests are reported in the following subsection.

E.2 PARAMETRIC TESTS

Additional tests of algorithm performance were made using the file BIGONE.SNR for the 34-node example network and node pair (25, 20). The SNRs in these files for a fixed jamming situation were made into a simulated parametric variation of jammer power by the method explained in Section 2.3.2.1 of the text.

E.2.1 **Algorithm Performance Comparisons**

Having eliminated the program EL2ONLY from consideration, as well as the programs that compute the exact value of s-t reliability, runs were made on a 386 computer to compare the performances of the following five algorithms in calculating $\gamma_{25,20}$ for the 34-node example network:

TABLE E-4 RESULTS FOR 34-NODE EXAMPLE, (3, 11)

Program	LB	UB	UB – LB	Time (sec)
EQLNKTST	.99999991 (98 successes)	1.00000000 (2 failures)	0.00	6.1
EL1&2	.99999990 (64 successes)	1.00000000 (36 failures)	0.00	7.2
EL2ONLY	----	-----	----	$\gg$ 4 minutes
ELCUTPAT	.99999928 (83 successes)	1.00000000 (23 failures)	0.00	9.3
TCPTRBND	.91294729	1.00000000	0.0871	0.5
RNPBOUND	.99998869	1.00000000	0.00	1

TABLE E-5 RESULTS FOR 34-NODE EXAMPLE, (18, 25)

Program	LB	UB	UB – LB	Time (sec)
EQLNKTST	.99995848 (100 successes)	1.00000000 (1 failure)	0.00	7
EL1&2	.99995646 (72 successes)	1.00000000 (28 failures)	0.00	7.7
EL2ONLY	----	-----	----	$\gg$ 4 minutes
ELCUTPAT	.99882084 (42 successes)	1.00000000 (72 failures)	0.0012	11.7
TCPTRBND	.93830301	1.00000000	0.0617	3.5
RNPBOUND	.99999791	1.00000000	0.00	3.5

TABLE E-6 RESULTS FOR 34-NODE EXAMPLE, (25, 20)

Program	LB	UB	UB – LB	Time (sec)
EQLNKTST	.99250576 (351 successes)	0.99952630 (18 failures)	0.0070	22.7
EL1&2	.99250576 (351 successes)	0.99937379 (34 failures)	0.0069	25.6
EL2ONLY	----	-----	----	$\gg$ 4 minutes
ELCUTPAT	.99098649 (883 successes)	0.99936825 (616 failures)	0.0084	115
TCPTRBND	.92678123	1.00000000	0.0732	1
RNPBOUND	.99897902	0.99937369	0.0004	10

Program	Algorithm implemented
EQLNKTST	Original ELA, with $\epsilon = 0.01$ and 1-way pathfinding using anti-pingpong (one-hop return prevention logic)
EL1&2	Both ELA and ELA2, using the lower bound from ELA and the upper bound from ELA2, using $\epsilon = 0.01$, 1-way pathfinding with anti-pingpong, and the selection of the larger of forward and reverse cutsets
ELCUTPAT	An algorithm that finds a path or a cutset, depending upon which generates the fewer new events, using $\epsilon = 0.01$, 1-way pathfinding with anti-pingpong, and selection of the larger of forward and reverse cutsets
TCPTRBND	Theologou-Carlier algorithm run twice: once to obtain an upper bound with the link reliability threshold $p_{th} = 0.9$, and once to obtain a lower bound with $p_{th} = 0.96$
RNPBOUND	Reduction and Partition algorithm with Theologou-Carlier approach to handle imperfect nodes; run with adaptive threshold once to obtain both upper and lower bounds.

The operation of these programs has been discussed previously, except for the particular heuristic adaptive link probability threshold in RNPBOUND that was used for this set of results. The adaptive threshold used may be expressed by

$$p_{th} = \beta_{max} - k \cdot (\beta_{max} - \overline{\beta})/15\,, \tag{E-2}$$

where k is an integer indexing the depth of recursion. That is, when the program first calls the probability calculation procedure, $k = 0$; each time that procedure calls itself in order to carry out the network factoring, k is incremented, causing the probability threshold to decrease. This thresholding scheme, used for both the lower and upper bound calculations, forces at least one recursion since for $k = 0$ none of the link reliabilities exceeds the threshold. For $k > 0$, factoring on a link with high reliability can result in an approximation—leading either to an upper bound or to a lower bound, as discussed in Section 2.2.3.2, by neglecting the probability of the subgraph with the factored link deleted. As the depth of the recursion increases, it is more likely that an approximation is used. By this heuristic method, the depth of recursion is made self-limiting and the overall execution time of the algorithm is contained.

It has been noted above in connection with the TCPTRBND program that the lower bound developed by the Theologou-Carlier type of algorithm tends to be loose; this tendency is especially pronounced for SNR conditions giving rise to sets of relatively low

link reliability values. The reason for the looseness is that the neglected term has been neglected on the basis that it is multiplied by $(1 - p_l)$, which is small for $p_l \approx 1$. However, for low SNR conditions, p_l is not close to 1. Therefore, in addition to implementing the adaptive threshold given in (E-2), the version of RNPBOUND used for the parametric performance results that follow employs an additional heuristic: the lower bound is computed as

$$\underbrace{\gamma_{st}(G) \geq p_l \cdot \gamma_{st}(G^{*}l)}_{\text{LB as expressed previously}} + \underbrace{(1 - p_l) \cdot \Pr\{\text{shortest } s \rightarrow t \text{ path}\}}_{\text{additional term to tighten LB}} \qquad \text{(E-3)}$$

Due to excessive run times experienced with the EL1&2 program, a 2400-second (40-minute) limit was implemented in the version of the program that was used to obtain the comparisons that follow.

Figure E-1 shows the execution time of the several programs as a function of the SNR threshold. Perhaps the most striking feature of the figure is the obvious time-limited behavior of the EL1&2 program for threshold values 9 to 12, despite the fact that this program has the lowest time for lower thresholds. It is difficult to understand how a 1 dB change could affect the performance of the program so drastically, yet these results appear to be correct. A close inspection of the operation of the program for a 9 dB threshold seemed to indicate that the program spent a disproportionate share of time in the earlier part of the calculation on servicing the queue containing events generated by the pathfinding or ELA portion of the algorithm; because of the relatively large hop distance for the selected (s, t) pair, each path that is found generates a relatively large number of next events that are put into the queue. The cutsets during the early part of the run, however, are relatively large (have few link outages), and the queue of next events generated by the ELA2 portion of the algorithm is smaller at first. However, accumulation of the upper bound is postponed until after the events contributing to the lower bound have been processed. Evidently, when the threshold is 8 dB, the program fortuitously achieves convergence before the queue sizes have become large.

In this light, it is interesting that the ELCUTPAT results (labelled “Cut Path" in Figure E-1) exhibit a more continuous increase in execution time for thresholds less than 18 dB, the point at which the topology of the surviving links in the network begins to simplify. Besides being faster than EL1&2 by virtue of maintaining one instead of two queues, ELCUTPAT seems to have a better performance by virtue of its event-minimizing strategy. However, as Figure E-1 shows, neither EL1&2 or ELCUTPAT is even close to the efficiency of the other three programs.

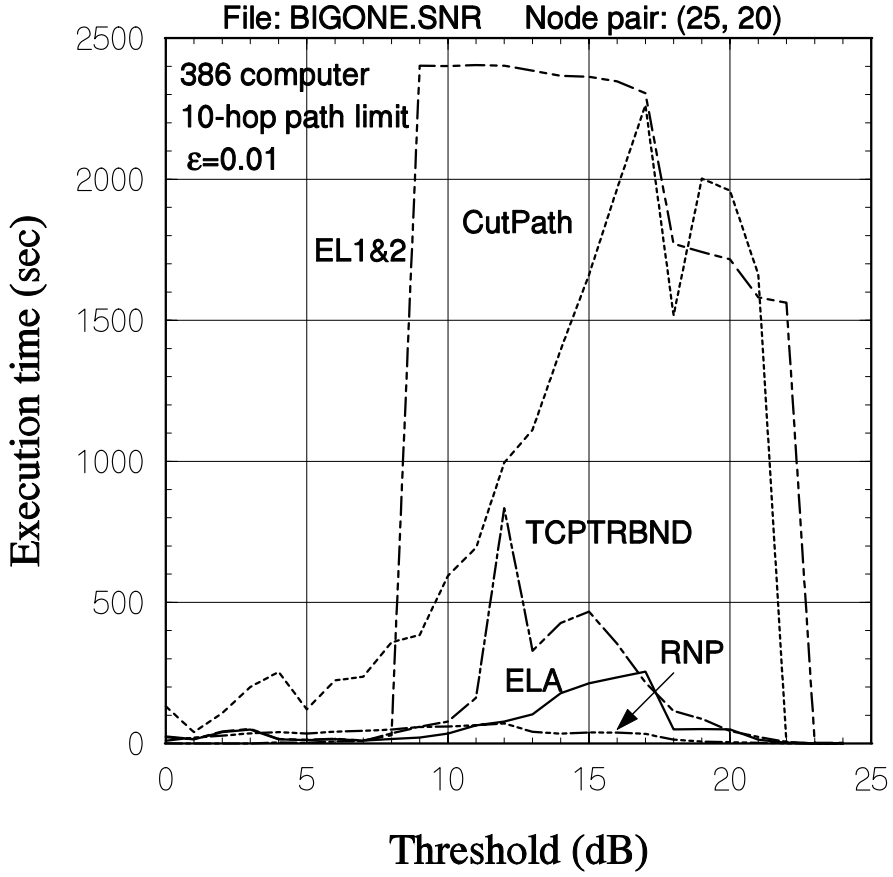


FIGURE E-1 COMPARISON OF EXECUTION TIMES

Figure E-2 presents an expanded view of Figure E-1 in which it is evident that none of the programs EQLNKTST (labelled “ELA"), TCPTRBND, and RNPBOUND (labelled “RNP") is consistently better than the others in terms of execution time. However, on the whole RNPBOUND performs the best, with EQLNKTST next in terms of speed. The execution time of TCPTRBND grows steeply for thresholds greater than 10 dB, due to the fact that fewer links exceed the fixed probability thresholds, then is sharply decreased for a threshold of 13 dB because the network has fewer UP links.

A comparison of the programs' accuracies is presented in Figure E-3. Again, the program EL1&2 is seen to be the worst in performance, even though it has the same value of the convergence stopping criterion, ϵ. The reason is that this program, for threshold values of 9 to 22 dB is stopping on the basis of other criteria: execution time (9 to 12 dB) and number of events (13 to 22 dB); a maximum of 10,000 events is one of the stopping criteria for the equivalent-links set of programs. Evidently the program EL-CUTPAT, while maintaining 0.01 accuracy for most of the threshold values, reaches the

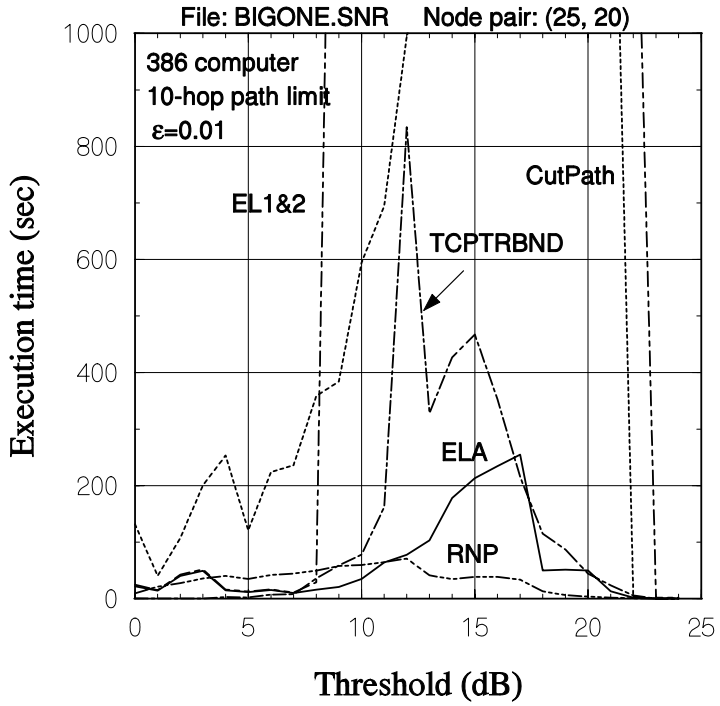


FIGURE E-2 EXPANDED VIEW OF FIGURE E-1

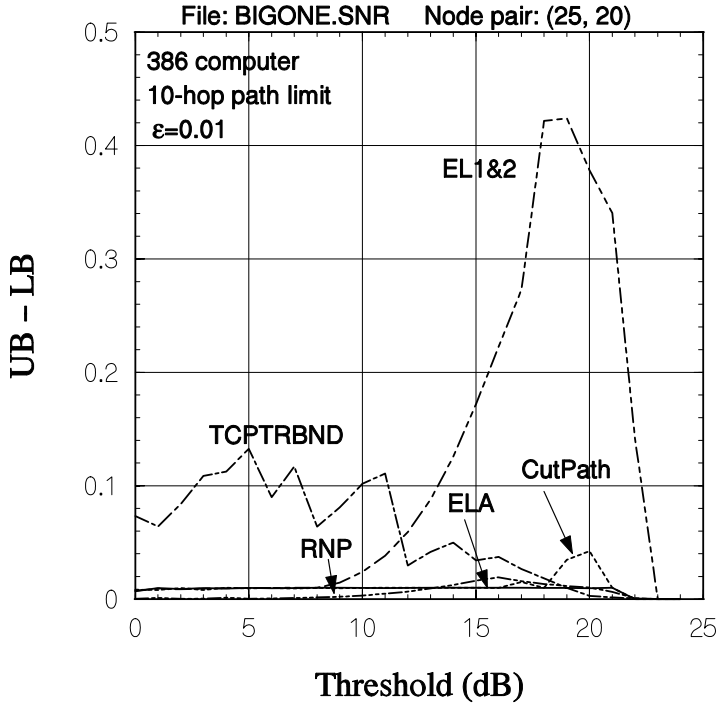


FIGURE E-3 COMPARISON OF ACCURACIES

maximum number events limit for threshold values of 16 and 18 to 20 dB, when the distance between its computed upper and lower bounds is greater than 0.01.

The program TCPTRBND has a somewhat variable accuracy. The parameters that determined the tightness of the bounds, the probability thresholds, are fixed and somewhat arbitrary, and the program could be made to attain a greater accuracy at the expense of execution time by raising those thresholds.

As for the comparison with respect to speed, therefore, the most promising programs appear to be EQLNKTST and RNPBOUND. Despite not having a parameter directly determining the distance between the bounds, RNPBOUND actually achieves tighter bounds for some values of the threshold, due to its adaptive probability threshold—in particular its forcing of at least one recursion per partitioned subgraph.

E.2.2 Comparison of *s-t* Reliability Estimates

One of the motives for considering the use of s-t cutsets instead of, or in combination with, s-t paths was the possibility of estimating the node-pair reliability by the arithmetic average of the upper and lower bounds, based on developing an algorithm that yields bounds that are equally accurate (or nearly so). The program EL1&2 is a prototype implementation of such an algorithm, using the lower bound from pathfinding success events and the upper bound from cutset-finding failure events. The program ELCUTPAT implements a different concept: the overall convergence of the upper and lower bounds to within ϵ can be done using fewer, larger events if the choice of path-finding or cutset-finding is made on the basis of minimizing the proliferation of network events. The accuracies of the bounds developed by these programs have been compared in the previous paragraphs, and now their performance in terms of estimating the network reliability will be examined.

As a basis of comparison, the criterion $\epsilon = 0.1$ was selected, representing a rather loose requirement on convergence, the idea being that it is fair to compare the estimation performance of the different programs when their bounds have achieved the same tolerance. Figure E-4 shows the execution times of the programs EQLNKTST, ELCUTPAT, and EL1&2. For EL1&2, an upper limit of 1500 sec was imposed to control the time required to obtain the data for this comparison; for this program and threshold values 18 to 21 dB, although less than 1500 sec the time was determined by the fact that the program halted due to the queue's filling up the available hard disk space (about 2.5 Mbytes). Thus the timing comparison for these programs for $\epsilon = 0.1$ is

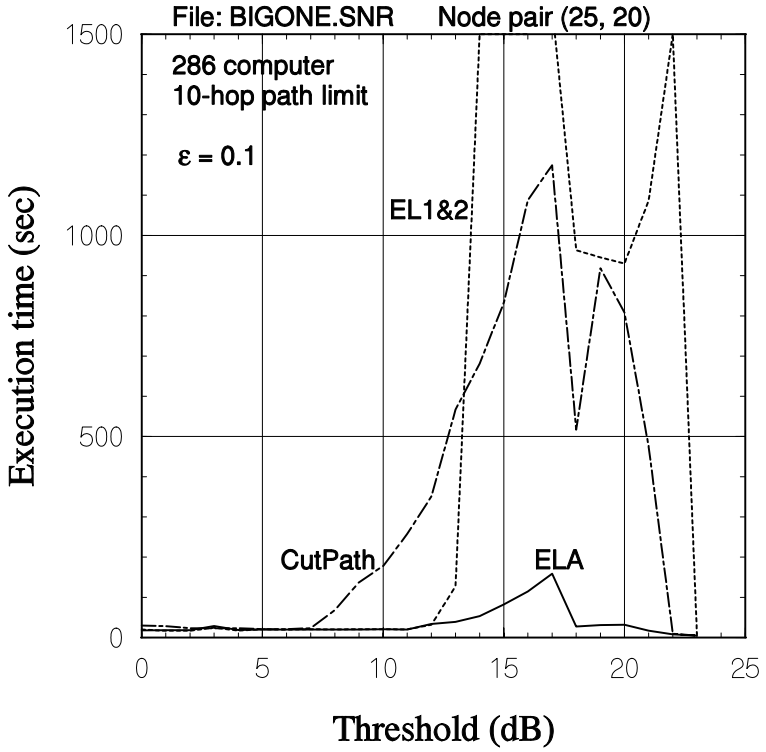


FIGURE E-4 COMPARISON OF TIMES FOR $\epsilon = 0.1$

consistent with that shown previously for $\epsilon = 0.01$ in Figure E-1: the program implementing the ELA maintains a relatively constant time except for thresholds between 12 and 17 dB; ELCUTPAT features a steadily increasing execution time until the network is simplified considerably at 18 dB; and EL1&2 does very well in terms of time for thresholds below a certain value, then very poorly.

The bias in the estimates generated by these programs are compared in Figure E-5, using the arithmetic average of the bounds produced by the ELA for $\epsilon = 0.01$ as the reference or "true" value of $\gamma_{25,20}$. As might be expected, all three of the programs underestimate the reliability for lower threshold values, since the true value (and upper bound) are close to 1.0 but the lower bound only has to exceed 0.9 to make UB – LB $< \epsilon$. For larger threshold values, when the lower bound tends to be tighter than the upper bound, all three programs overestimate the reliability; the lower bound is tighter because the success events, though fewer and accumulated at various time positions as the different algorithms proceed, generally have higher probabilities. When it is working efficiently, the EL1&2 program bias behavior parallels that of the EQLNKTST program,

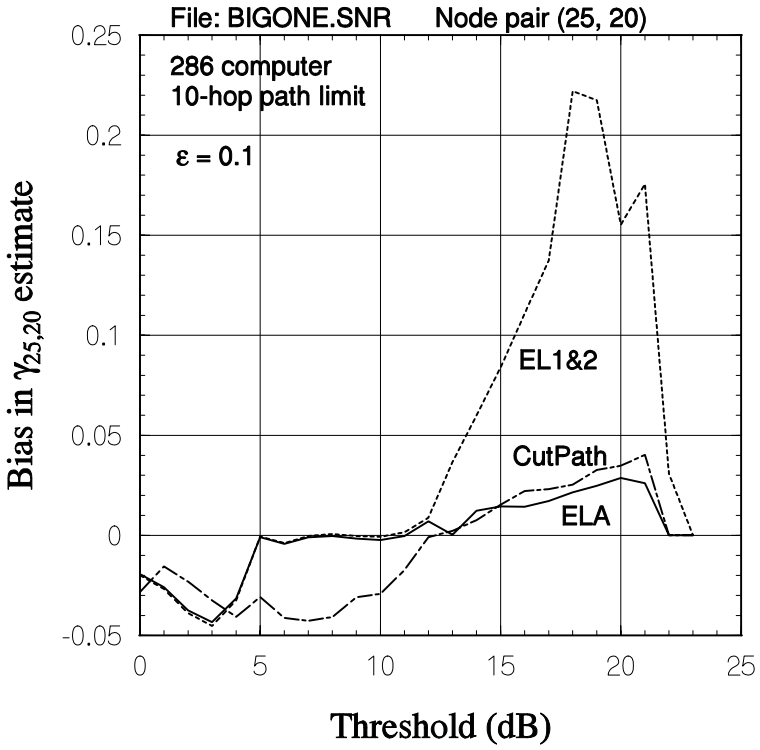


FIGURE E-5 COMPARISON OF ESTIMATES FOR $\epsilon = 0.1$

and seems to produce as good or better an estimate, but the ELCUTPAT has a different bias behavior altogether.

Additional data was collected for EL1&2 and ELCUTPAT with an execution time limit of 160 seconds, the maximum value experienced by the EQLNKTST program, to see if somehow the intermediate values of the bounds produced by these programs have a smaller bias. Thus time-limited runs were made for ELCUTPAT for 10 to 21 dB thresholds, with the result that the new bias values were generally greater in magnitude; there was a smaller bias for 15 and 16 dB, where the bias is changing from negative to positive. Time-limited runs for the EL1&2 program for 14 to 22 dB thresholds yielded the same results: a slightly smaller bias for 15 and 16 dB, but a larger bias otherwise.

E.2.3 **Effects of variations in the path search method**

Using insights gained from the results described above, further parametric tests of the *s-t* reliability programs were conducted for the case of the 34-node example network. These tests were designed (1) to determine the effect on execution time of variations in

the path and cutset search methods used by the ELA family of algorithms, and (2) to determine the performance of the TCPTRBND program when it is modified to use the same adaptive threshold techniques as the RNPBOUND program.

In order to study the effect of variations in the path search method, network reliability calculations using the ELA (program EQLNKTST and modifications to it) were made on the 286 computer for the four possible combinations of the following two path search techniques: the search propagation technique and the search redundancy control technique. In addition, a concentrated effort was made to optimize the program for the search in terms of execution time.

- *Search propagation technique*: previous runs utilized a one-way "flooding" forward from the source node (s) in search of the destination node (t). That is, in steps corresponding to transmission hops, information on the links emanating from the nodes so far reached on step i is used to determine the nodes that would be reached on hop $i+1$.[15] The search stops when t is reached, or when the maximum number of hops have been used without reaching t. An alternative search propagation method is to alternate between flooding forward from s and backward from t; in this manner a path (if one exists) is discovered if the set of nodes reached in the forward direction has a node in common with the set of nodes in the backward direction.[16] Although it takes additional logic to implement the two-way search, there is a potential for reductions on the order of 50% in the search time for cases in which s and t are widely separated.

The basis of this advantage may be explained intuitively as follows: the flood search may be thought of as having a radius in terms of the number of hops from the source of the search. Thus a one-way search from s to t that finds a path of length L hops covers an "area" with radius L hops. A two-way search presumably would stop after forward and backward searches had each covered areas with radius $L/2$ hops. The number of nodes included in a given hop radius obviously depends upon the properties of the network—for a fully connected network, all the network nodes are included in a one-hop radius from any node, while for an infinite mesh network, there are $4L(L+1)$ nodes in a radius of L hops. Thus the ratio of the number of nodes reached by a two-way path search to that reached by a one-way search can be modelled for a mesh network by

[15]The convention used by the program for a search in a given direction is that the first time that a node is reach on a given hop, the connection thus established is not replaced by subsequent connections that are found to be possible for the hop as the program executes its "loops" using the node number (label) as an index. That is, preference is given to the "lexicographically first" connection.

[16]The "lexicographically first" connections in the half of the path search performed by the backward search are sometimes different from those that are found by a one-way search, giving rise to the finding of different paths—sometimes more advantageous to the execution time, sometimes not.

$$\frac{2 \times \#\text{ nodes in radius } L/2}{\#\text{ nodes in radius } L} = \frac{8(L/2)(L/2+1)}{4L(L+1)} = \frac{L/2+1}{L+1} . \qquad \text{(E-4)}$$

For large L, obviously the ratio approaches 1/2.

- *Search redundancy control technique*: the path search method employed to obtain the results presented in this report so far featured what was called “anti-pingpong" logic to prevent the propagation of the search to a node visited on the previous hop. For example, propagation from the nodes reached on the first hop forward from s are not allowed to propagate backwards to s on the second hop. This logic helps to prevent the waste of computation time that would result if the flood search continually regenerated itself by going in loops. However, although this logic prevents the return to a previously visited node two hops later, it does not prevent the return to a previously visited node in *more* than two hops later. Therefore, “anti-return" logic was developed to prevent *any* returns to previously visited nodes, simply by remembering the nodes so far reached by the search and by not permitting the search to propagate to any of them.

The effects of these variations in the path search technique on the program execution time and on the numbers of events are demonstrated in Figures E-6 and E-7, respectively. It may be observed from Figure E-6 that an overall reduction in the time of more than 50% is experienced for certain threshold values, and that generally going from anti-pingpong logic to anti-return logic reduces the time more than going from a one-way search to a two-way search. The time is seen to be very dependent upon the numbers of success and failure events that are processed, which are different for the two search propagation techniques, as illustrated in Figure E-7, due to a necessarily different lexicographical ordering of preferred paths in the subroutine that implements the two-way search. Evidently in this example the labelling of the nodes and links in conjunction with the order in which the links fail as the threshold increases is such that the one-way search had a lexicographical advantage for low values of the threshold, but had a disadvantage for higher values.

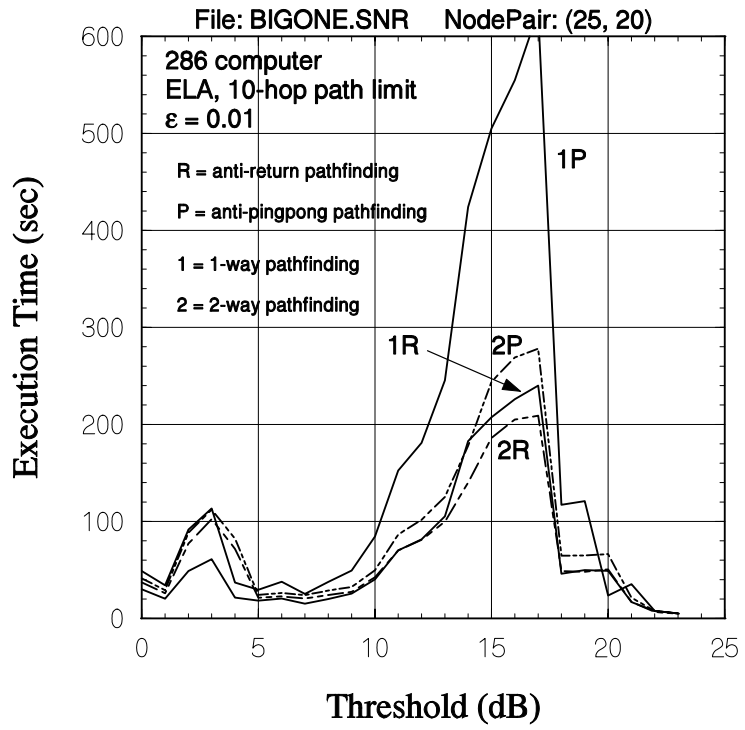


FIGURE E-6 EFFECT OF PATH SEARCH METHOD ON TIME

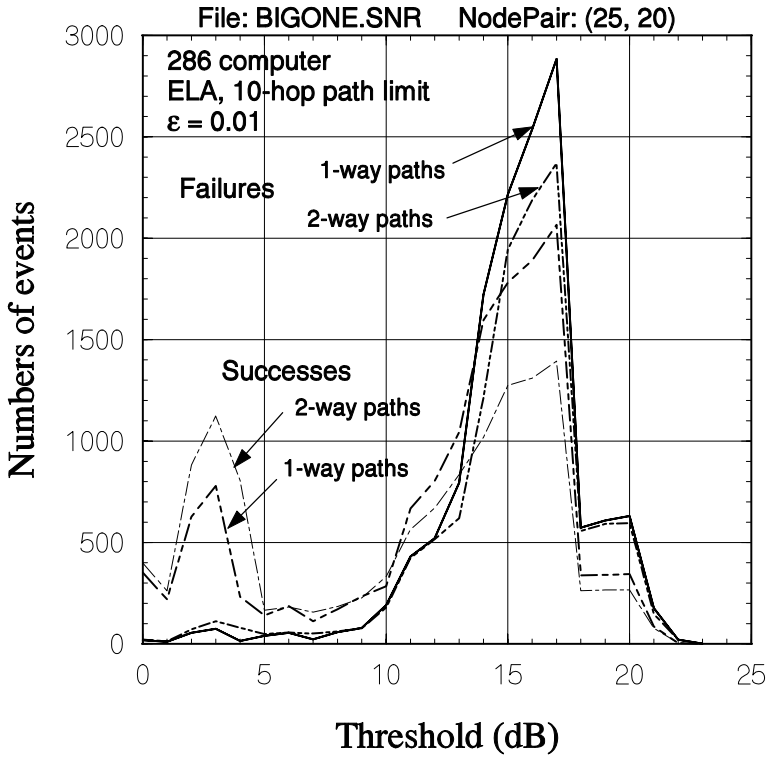


FIGURE E-7 EFFECT OF PATH SEARCH METHOD ON EVENTS

- *Optimization of the program.* Further reductions in the program execution time were found to be possible by more efficient programming. For example, common to the path search methods is the requirement to initialize an $N \times N$ matrix; setting (assigning) the individual elements of one row to zero and then copying that row $N - 1$ times (using the TurboPascal *Move* procedure to copy the appropriate number of bytes) turns out to be significantly faster than doing the same work by assignment statements. Also, when it is desired simply to check whether a node has been reached, it is faster to keep track of the nodes reached in an $N \times 1$ vector whose elements are zero for nodes not reached and one, otherwise—as opposed to using the Pascal data structures and operations for sets to check whether a given node is in the set of nodes reached.

The curves shown before in Figure E-6 are repeated in Figure E-8, which includes also “optimized" execution time results for the one- and two-way path search methods with anti-return logic. The effort to optimize the path search procedures and functions in the NETSET Pascal unit are seen to have paid off in an overall reduction in the execution time of up to 75% for the particular network example.

E.2.4 **Effect of variations in the cutset search method**

In order to study the effect of variations in the cutset search method, network reliability calculations using the ELA1&2 program (and modifications to it) were made on the 286 computer for the eight possible combinations of the two path search techniques and the selection of either of two variations on the cutset search method.

- *Larger cutset method*. The results shown previously in this report for the programs that involve searching for cutsets were obtained using the cutset search procedure that was illustrated in Figure 1-4. This procedure finds both forward and reverse cutsets, and selects the larger of the two (the one that specifies fewer required link outages) in order to minimize the number of new events that will be generated.

- *First cutset method*. A feature of the original cutset search concept [4] was that utilization of a two-way search for cutsets would be a fast technique for finding a single cutset, not a method for obtaining two cutsets. It has been hypothesized that the speed of selecting the first cutset found will be more influential in saving execution time than will the selection of a larger cutset.

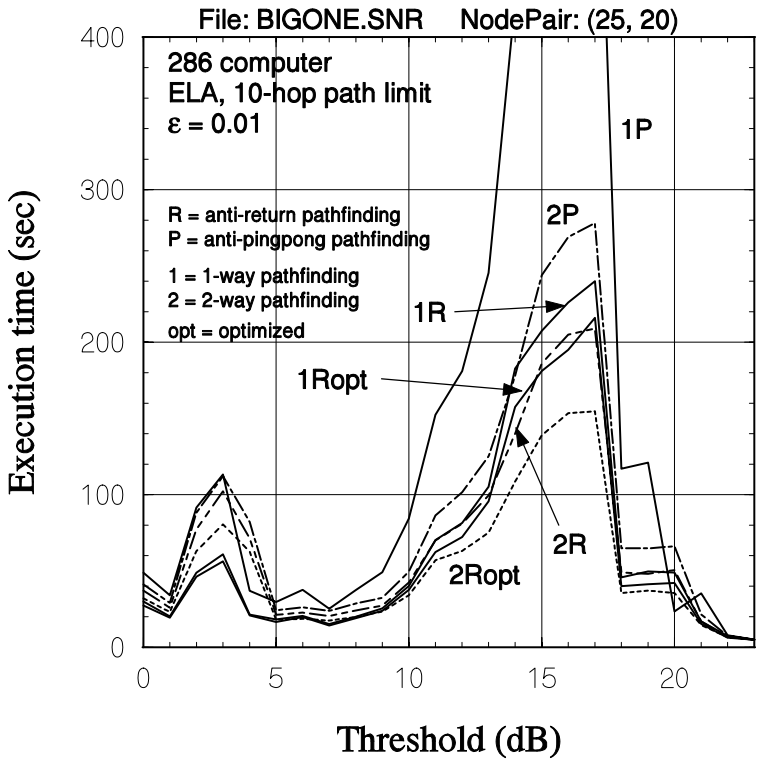


FIGURE E-8 EFFECT OF OPTIMIZATION ON ELA PROGRAM TIME

Prior to the implementation of more efficient path search programming, the results of testing the performance of ELA1&2 for variations in path and cutset search methods are presented in Figures E-9 to E-11. The salient feature of Figure E-9 is its disclosure that the execution time of the ELA1&2 program is more influenced in this case by the choice of path search technique than it is by the choice of cutset search technique, and that time reductions on the order of 40% are experienced.[17] As before, the one-way path search with anti-return logic is uniformly better than the two-way path search with anti-return logic for the threshold values shown. The use of the first cutset found generally does not affect the time significantly, and is neither uniformly better nor uniformly worse than the use of the larger cutset.

The reason for this behavior is apparent from comparing Figure E-9 with Figures E-10 and E-11 and observing that the number of success events (which is not affected by the cutset search method in this case) largely determines the execution time; as noted previously, the lexicographical ordering of nodes and links is such that the one-way path

[17]Note that the ELA1&2 program still requires excessive time for threshold values greater than 8 dB; results for the higher threshold values are shown below for more efficient programming.

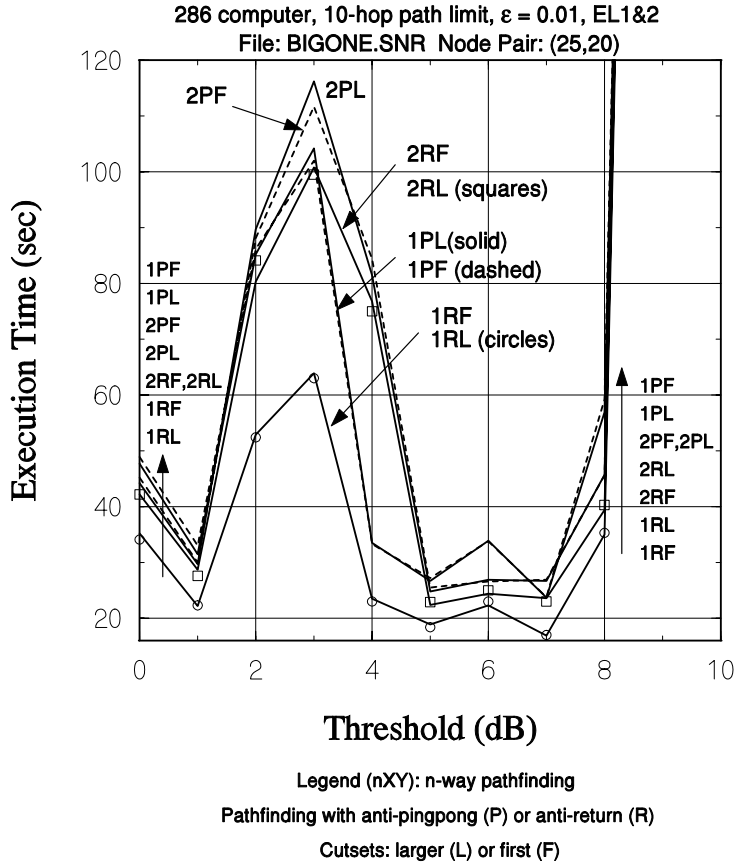


FIGURE E-9 EFFECT OF VARIATIONS ON EL1&2 PROGRAM TIME

search produces fewer (larger) success events than does the two-way search. Note that the number of failure events is affected by the path search method as well as by the cutset search method; the path search method affects the rate at which the lower bound accumulates, which in turn affects the number of failure events that must be processed to achieve upper and lower bound convergence to within $\epsilon = 0.01$. The use of the larger cutset is shown in Figure E-11 to result in a much as a 20% reduction in the number of failure events needed to achieve bound convergence; however, it is clear from the figure that this improvement rarely occurs, and has little effect on the time.

- *Results after program optimization.* In addition to speeding up the path search program as discussed above, an effort was made to improve the speed of the cutset search procedures and functions, using some of the same programming techniques. Also, it was discovered that the portion of the time *vs.* threshold curves for which the EL1&2 program time was excessive in the results presented above can be characterized as follows: the EL1 part of the program has completed its enumeration of all the success events, yielding

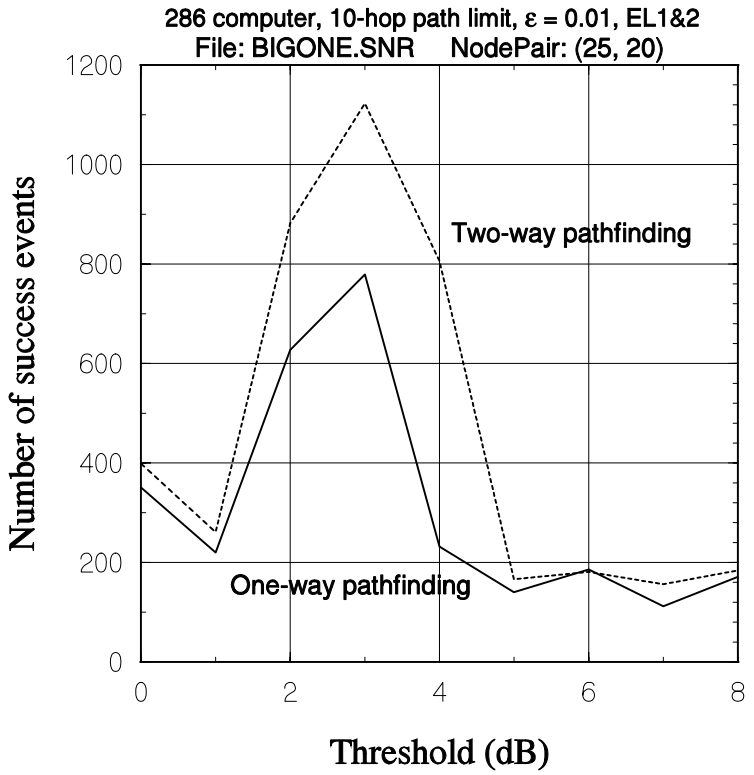


FIGURE E-10 EFFECT OF VARIATIONS ON EL1&2 SUCCESS EVENTS

a lower bound that is actually an exact computation of the *s-t* reliability, while the EL2 part of the program is still processing the thousands of cutset-generated events that are associated with the particular network example. Based on this discovery, the logic of the procedures in the CUTSET Pascal unit was modified to accomplish the following objectives:

(1) Detect when either part of the program has found the exact answer (by noting when the queue is not added to), and stop the program at that point.

(2) In this eventuality, cause the estimate to be set equal to the exact answer instead of the average of the upper and lower bounds.

The optimized EL1&2 program produced the results illustrated in Figure E-12, which includes the optimized results for the ELA as well. This figure may be compared directly with Figures E-1 and E-2, rather than with Figure E-9, since the 386 computer was used to obtain them. Note that the various improvements in the EL1&2 program have reduced the worst-case execution time by almost an order of magnitude. Note also

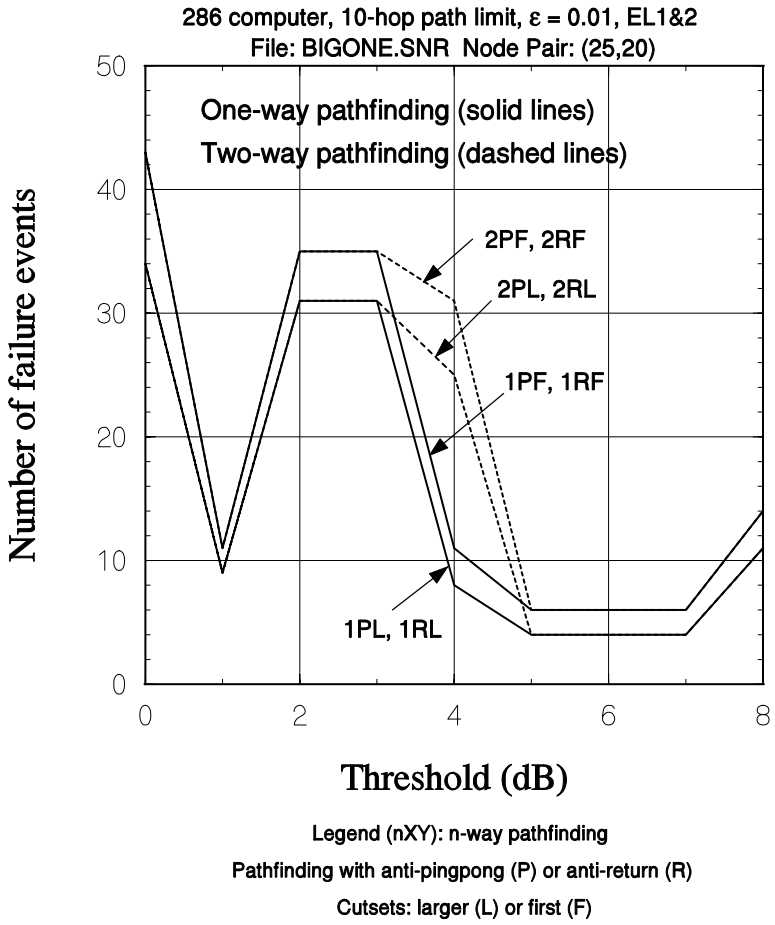


FIGURE E-11 EFFECT OF VARIATIONS ON EL1&2 FAILURE EVENTS

that the two-way pathfinding method does outperform the one-way method for threshold values greater than 8 dB.

Optimized results for the ELCUTPAT program, which uses both paths and cutsets, are shown in Figure E-13. As discussed in Section 2.2.2.4, according to the original program concept, events are processed either by pathfinding (in search of a success event) or by a cutset search (to find a failure event), with preference given to the method that yields the fewer number of new events to be placed in the queue. Figure E-13 shows the effect of giving a slightly stronger preference to paths, by selecting a path rather than a cutset if either the path generates fewer events OR if the path generates two or fewer new events (when conceivably a cutset might generate only one new event). Evidently, since the upper and lower bound calculations (not shown) were not affected, the approximately 10% time savings that is displayed in Figure E-13 (somewhat more for two-way paths)

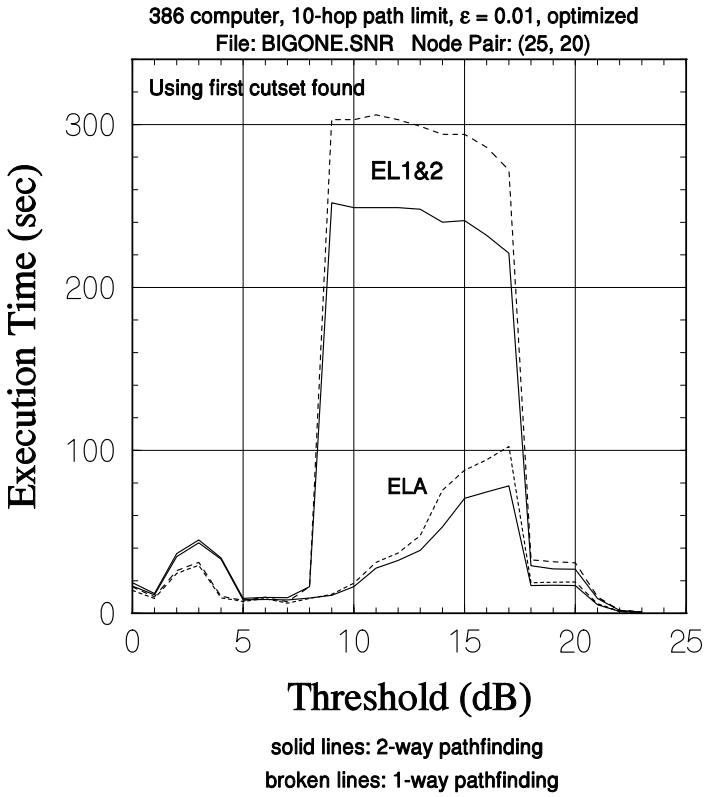


FIGURE E-12 EFFECT OF OPTIMIZATION ON EL1&2 TIME

when the presumptive selection criterion is $NewCount \leq 2$ is due entirely to not executing a cutset search when $NewCount = 2$.

If an increasing preference is given to paths, eventually the algorithm reverts to the ELA, which only uses pathfinding. It may be the case that for some level of preference for paths, short of a total preference, the algorithm succeeds in its objective of improving upon the ELA in terms of execution time because of a more efficient partitioning of events. However, this circumstance seemed to be very unlikely, so the effort was not made to obtain results for a full variation of the path/cutset selection criterion.

A comparison of Figure E-13 with Figure E-1 reveals that the optimization of the searches improved the performance of ELCUTPAT by better than 60% for thresholds up to 17 dB. It is interesting that the use of one-way pathfinding by this program resulted (except for a 5 dB threshold) in a uniformly smaller execution time than for two-way pathfinding. It is anticipated that the better performance of two-way pathfinding for higher threshold values that has been observed for the ELA would also be the case for ELCUTPAT at some point when there is a stronger preference for paths.

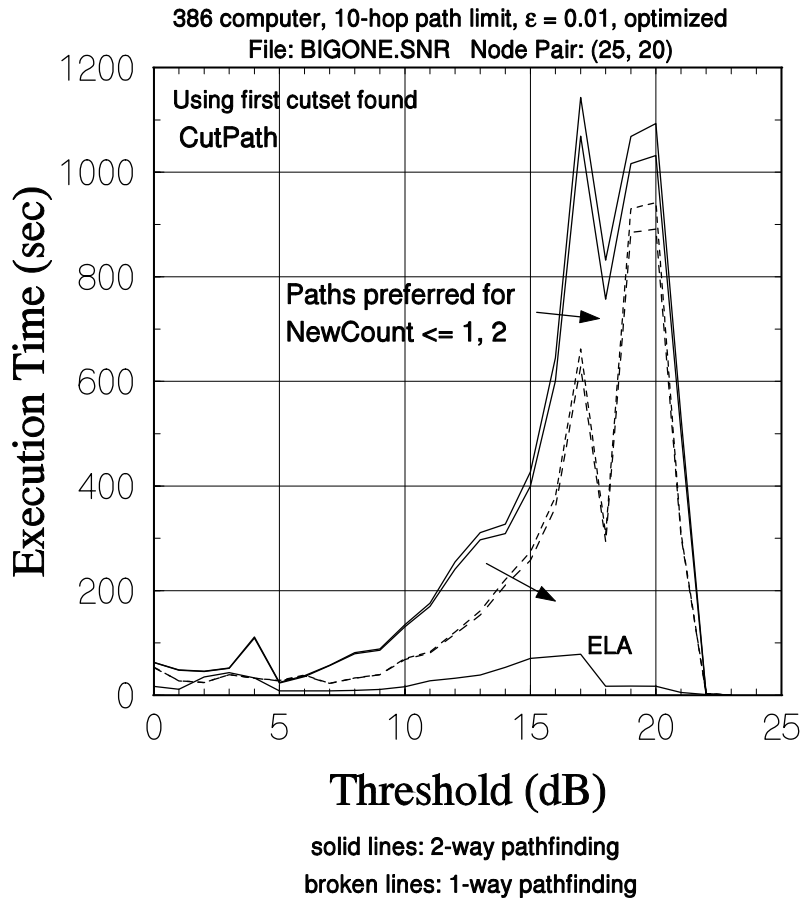


FIGURE E-13 EFFECT OF OPTIMIZATION ON ELCUTPAT TIME

REFERENCES

[1] L. E. Miller and J. J. Kelleher, “Analysis Models for MSE Network Survivability Performance in the Tactical Environment," J. S. Lee Associates, Inc. report JC-2091-7-FF, March 1992 (DTIC accession number AD-B164578).

[2] D. J. Torrieri, “An Efficient Algorithm for the Calculation of Node-Pair Reliability," *Proc. IEEE 1991 Milit. Comm. Conf.* (MILCOM '91), McLean, VA, 4-7 November, pp. 0187-0192.

[3] W. P. Dotson and J. O. Gobien, “A New Analysis Technique for Probabilistic Graphs," *IEEE Trans. on Circuits and Syst.*, vol. 26, pp. 855-865, October 1979.

[4] D. J. Torrieri, “New Algorithm for Large Networks," notes dated 6 April 1992.

[5] L. E. Miller, L. Wong, and J. J. Kelleher, “Net Survivability Analysis Software Tools," J. S. Lee Associates, Inc. report JC-2092-3-F under contract DAAL02-89-C-0040, August 1992.

[6] L. E. Miller, “Propagation Model Sensitivity Study," J. S. Lee Associates, Inc. report JC-2092-1-FF, July 1992 (DTIC accession number AD-B166479).

[7] L. E. Miller, J. J. Kelleher, and J. S. Lee, “Analysis of the Survivability of Large MSE Networks by a Truncated Equivalent-Links Methodology," *Proc. IEEE 1992 Milit. Comm. Conf.* (MILCOM '92), San Diego, CA, 12-14 October, pp. 0487-0491.

[8] L. E. Miller, “Army JTIDS Communications Survivability Modelling and Jammed Connectivity Assessment (U)" (Secret report), J. S. Lee Associates, Inc. report JC-2091-6-FF under contract DAAL02-89-C-0040, July 1992.

[9] L. B. Page and J. E. Perry, “Reliability of Directed Networks Using the Factoring Theorem," *IEEE Trans. on Reliability*, vol. R-38, pp. 556-562, December 1989.

[10] O. R. Theologou and J. G. Carlier, “Factoring and Reductions for Networks with Imperfect Vertices," *IEEE Trans. on Reliability*, vol. 40, pp. 210-217, June 1991.

[11] N. Deo and M. Medidi, “Parallel Algorithms for Terminal-Pair Reliability," *IEEE Trans. on Reliability*, vol. 41, pp. 201-209, June 1992.

[12] J. M. Wilson, “An Improved Minimizing Algorithm for Sum of Disjoint Products," *IEEE Trans. on Reliability*, vol. 39, pp. 42-45, April 1990.

[13] L. E. Miller, J. J. Kelleher, and L. Wong, "Assessment of Network Reliability Calculation Methods," J. S. Lee Associates, Inc. report JC-2097-FF under contract DAALo2-92-C-0045, January 1994 (DTIC accession number AD-B170336).